Università degli Studi di Bologna

FACOLTÀ DI INGEGNERIA

Dottorato di Ricerca in Ingegneria Elettronica,
Informatica e delle Telecomunicazioni

Ciclo XX

Tecniche per il controllo dinamico del consumo di potenza per piattaforme System-on-Chip

Tesi di Dottorato di:

Martino Ruggiero

Relatori :

Chiar. mo Prof. Ing. **Luca Benini**

Coordinatore:

Chiar. mo Prof. Ing. **Paolo Bassi**

Settore Scientifico Disciplinare: ING/INF01 Elettronica

Anno Accademico 2006/2007

TECNICHE PER IL CONTROLLO DINAMICO DEL CONSUMO DI POTENZA PER PIATTAFORME SYSTEM-ON-CHIP

A dissertation submitted to the
DEPARTEMENT OF ELECTRONICS, COMPUTER SCIENCE AND SYSTEMS
OF UNIVERSITY OF BOLOGNA

for the degree of Doctor of Philosophy

presented by
MARTINO RUGGIERO
born July 16, 1979

March 2008

Keywords

System-on-Chip (Soc)

Low-Power

MultiProcessor System-on-chip (MPSoC)

Power Management

Scheduling Algorithms

LCD

BackLight Scaling

Contents

List of Figures

List of Tables

Chapter 1

Abstract

Providing support for multimedia applications on low-power mobile devices remains a significant research challenge. This is primarily due to two reasons:

- Portable mobile devices have modest sizes and weights, and therefore inadequate resources, low CPU processing power, reduced display capabilities, limited memory and battery lifetimes as compared to desktop and laptop systems.

- On the other hand, multimedia applications tend to have distinctive QoS and processing requirements which make them extremely resource-demanding.

This innate conflict introduces key research challenges in the design of multimedia applications and device-level power optimization.

Energy efficiency in this kind of platforms can be achieved only via a synergistic hardware and software approach. In fact, while System-on-Chips are more and more programmable thus providing functional flexibility, hardware-only power reduction techniques cannot maintain consumption under acceptable bounds.

It is well understood both in research and industry that system configuration and management cannot be controlled efficiently only relying on low-level firmware and hardware drivers. In fact, at this level there is lack of information about user application activity and consequently about the impact of power management decision on QoS.

Even though operating system support and integration is a requirement for effective performance and energy management, more effective and QoS-sensitive power management is possible if power awareness and hardware configuration control strategies are tightly integrated with domain-specific middleware services.

The main objective of this PhD research has been the exploration and the integration of a middleware-centric energy management with applications and operating-system. We choose to focus on the CPU-memory and the video subsystems, since they are the most power-hungry components of an embedded system. A second main objective has been the definition and implementation of software facilities (like toolkits, API, and run-time engines) in order to improve programmability and performance efficiency of such platforms.

Enhancing energy efficiency and programmability of modern Multi-Processor System-on-Chips (MPSoCs)

Consumer applications are characterized by tight time-to-market constraints and extreme cost sensitivity. The software that runs on modern embedded systems must be high performance, real time, and even more important **low power**. Although much progress has been made on these problems, much remains to be done.

Multi-processor System-on-Chip (MPSoC) are increasingly popular platforms for high performance embedded applications. This leads to interesting challenges in software development since efficient software development is a major issue for MPSoc designers.

An important step in deploying applications on multiprocessors is to allocate and schedule concurrent tasks to the processing and communication resources of the platform. The problem of allocating and scheduling precedence-constrained tasks on processors in a distributed real-time system is NP-hard. There is a clear need for deployment technology that addresses these multi processing issues. This problem can be tackled by means of specific middleware which takes care of allocating and scheduling tasks on the different processing elements and which tries also to optimize the power consumption of the entire multiprocessor platform.

This dissertation is an attempt to develop insight into efficient, flexible and optimal methods for allocating and scheduling concurrent applications to multiprocessor architectures.

It is a well-known problem in literature: this kind of optimization problems are very complex even in much simplified variants, therefore most authors propose simplified models and heuristic approaches to solve it in reasonable time. Model simplification is often achieved by abstracting away platform implementation ”details”. As a result, optimization problems become more tractable, even reaching polynomial time complexity. Unfortunately, this approach creates an *abstraction gap* between the optimization model and the real HW-SW platform. The main issue with heuristic or, more in general, with incomplete search is that they introduce an *optimality gap* of unknown size. They provide very limited or no information on the distance between the best computed solution and the optimal one.

The goal of this work is to address both abstraction and optimality gaps, formulating accurate models which accounts for a number of ”non-idealities” in real-life hardware platforms, developing novel mapping algorithms that deterministically find optimal solutions, and implementing software infrastructures required by developers to deploy applications for the target MPSoC platforms.

Energy Efficient LCD Backlight Autoregulation on Real-Life Multimedia Application Processor

Despite the ever increasing advances in Liquid Crystal Display’s (LCD) technology, their power consumption is still one of the major limitations to the battery life of mobile appliances such as smart phones, portable media players, gaming and navigation devices. There is a clear trend towards the increase of LCD size to exploit the multimedia capabilities of portable devices that can receive and render high definition video and pictures. Multimedia applications running on these devices require LCD screen sizes of 2.2 to 3.5 inches and more to display video sequences and pictures with the required quality.

LCD power consumption is dependent on the backlight and pixel matrix driving circuits and is typically proportional to the panel area. As a result, the contribution is also likely to be considerable in future mobile appliances. To address this issue, companies are proposing low power technologies suitable for mobile applications supporting low power states and image control techniques.

On the research side, several power saving schemes and algorithms can be found in literature. Some of them exploit software-only techniques to change the image content to reduce the power associated with the crystal polarization, some others are aimed at decreasing the backlight level while compensating the luminance reduction by compensating the user perceived quality degradation using pixel-by-pixel image processing algorithms. The major limitation of these techniques is that they rely on the CPU to perform pixel-based manipulations and their impact on CPU utilization and power consumption has not been assessed.

This PhD dissertation shows an alternative approach that exploits in a smart and efficient way the hardware image processing unit almost integrated in every current multimedia application processors to implement a hardware assisted image compensation that allows dynamic scaling of the backlight with a negligible impact on QoS. The proposed approach overcomes CPU-intensive techniques by saving system power without requiring either a dedicated dis-

play technology or hardware modification.

Thesis Overview

The remainder of the thesis is organized as follows.

The first part is focused on enhancing energy efficiency and programmability of modern Multi-Processor System-on-Chips (MPSoCs). Chapter 2 gives an overview about architectural trends in embedded systems, illustrating the principal features of new technologies and the key challenges still open. Chapter 3 presents a QoS-driven methodology for optimal allocation and frequency selection for MPSoCs. The methodology is based on functional simulation and full system power estimation. Chapter 4 targets allocation and scheduling of pipelined stream-oriented applications on top of distributed memory architectures with messaging support. We tackled the complexity of the problem by means of decomposition and no-good generation, and prove the increased computational efficiency of this approach with respect to traditional ones. Chapter 5 presents a cooperative framework to solve the allocation, scheduling and voltage/frequency selection problem to optimality for energy-efficient MPSoCs, while in Chapter 6 applications with conditional task graph are taken into account. Finally Chapter 7 proposes a complete framework, called Cellflow, to help programmers in efficient software implementation on a real architecture, the Cell Broadband Engine processor.

The second part is focused on energy efficient software techniques for LCD displays. Chapter 8 gives an overview about portable device display technologies, illustrating the principal features of LCD video systems and the key challenges still open. Chapter 9 shows several energy efficient software techniques present in literature, while Chapter 10 illustrates in details our method for saving significant power in an LCD panel.

Finally, conclusions are drawn, reporting the main research contributions that have been discussed throughout this dissertation.

Chapter 2

Introduction

2.1 Trends in Embedded Systems: A Consumers Perspective

The number of consumer electronics devices sold worldwide is growing rapidly. A total of 2.1 billion consumer electronics devices with a total value of \$1.3 trillion were sold worldwide in 2006. It is expected that by 2010 this has grown to over 3 billion devices with a total value of around \$1.6 trillion [1]. Most of these devices contain one or more processors that are used to realize the functionality of the device. This type of devices are called embedded systems. Embedded systems range from portable devices such as digital cameras and MP3-players, to systems like a television or the systems controlling the flight of an airplane. These systems are everywhere around us in our daily live. Most of them are becoming intelligent micro-systems that interact with each other, and with people, through (wireless) sensors and actuators. Embedded systems form the basis of the so-called post-PC era [2], in which information processing is more and more moving away from just PCs to embedded systems. This trend is also signaled by ubiquitous computing [3], pervasive computing [4] and ambient intelligence [5]. These three visions describe all a world in which people are surrounded by networked embedded systems that are sensitive to their environment and that adapt to this environment. Their objective is to make information available anytime, anywhere. Embedded systems provide the necessary technology to realize these visions [6]. Realization of these visions implies that the number of embedded systems surrounding us in our daily lives will increase tremendously.

An important subclass of embedded systems are embedded multimedia systems. These systems combine multiple forms of information content and information processing (e.g. audio, video, animations, graphics) to inform or

Figure 2.1: Embedded multimedia systems: PlayStation 3 and iPhone.

entertain the user. Examples of such systems are mobile phones, game consoles, smart cameras and set-top boxes. Many of the applications that perform the information processing in these systems process audio, video and animation. These types of data are inherently streaming. So, many embedded multimedia systems contain streaming applications [7]. These applications typically perform a regular sequence of transformations on a large (or virtually infinite) sequence of data items.

The functionality integrated into new embedded multimedia systems is ever increasing. The Sony PlayStation has, for example, transformed itself from a simple game console to a complete entertainment center. It not only allows users to play games, it can also be used to watch movies, listen to music and to browse the Internet or chat online with other PlayStation 3 users. Another example of a true multimedia platform is the Apple iPhone. It includes many different applications next to the mobile-phone functionality. It has, for example, a wide-screen LCD display that allows users to watch movies and browse through their collection of photos that are taken with the build-in camera. The phone contains also an MP3-player which allows users to listen for up-to 16 hours to their favorite music. While traveling, users can also use the phone to browse the Internet, send emails or use online navigation software such as Google-maps. It is expected that even more functions will be integrated into future embedded multimedia systems. This trend was already signaled by Vaandrager in 1998 who stated that ”for many products in the area of consumer electronics the amount of code is doubling every two years" [8].

Current embedded multimedia systems have a robust behavior. Consider for example a modern high-end television system. Such a system splits the incoming video stream from its accompanying audio stream. Many different picture enhancement algorithms are executed on the video stream to improve its quality when displayed on the screen. Despite the complex processing going on inside the television, the video and audio stream are output in sync on the screen and the speakers. Consumers expect that future embedded multimedia systems provide the same robust behavior as current systems have despite the fact that more and more media processing is performed in software [9].

In summary, the following trends in embedded (multimedia) systems are observed from the perspective of consumers.

- The number of embedded systems surrounding people in their daily lives is growing rapidly, and these systems are becoming connected more and more often.
- Increasingly more functionality is integrated into a single multimedia system.
- Users expect the same seamless behavior of all functions offered by novel multimedia systems as offered by existing systems.

2.2 Trends in Embedded Systems: A Designers Perspective

The previous section outlines the most important trends in the field of embedded systems from the perspective of consumers. It shows that embedded systems have to handle an increasing number of applications that are concurrently executed on the system. At the same time, guarantees must be provided on the behavior of each application running on the system. This section considers the same systems, but it looks at the trends visible in their design(-process).

The omnipresence of embedded systems in people's lives is leading to a tremendous increase in the amount of data that is being used. Today, people have gigabytes of photos, music and video on their systems. That data must be processed in real-time to be made useful. Embedded systems must provide the required computational power to do this. At the same time, their energy consumption should be kept at a minimum as many of these devices are battery powered (e.g., mobile-phone, MP3-player, digital-camera). To fulfill these requirements, the use of multi-processor systems-on-chip (MP-SoCs) is becoming increasingly popular [10], [11]. For example, Intel has shifted from increas-

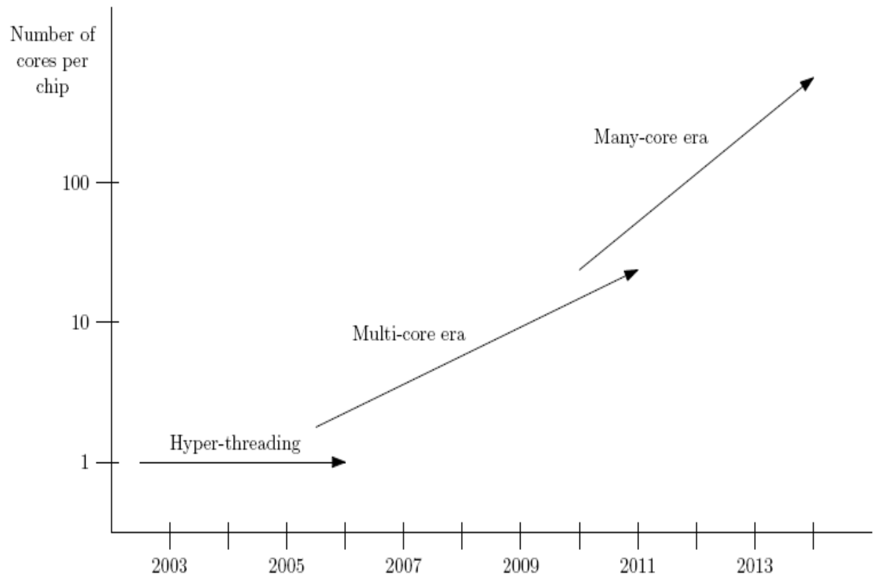


Figure 2.2: Current and expected eras for Intel processor architectures [12].

ing the clock frequency of every processor generation to a strategy in which multiple cores are integrated on a single chip. This paradigm shift is outlined in their platform 2015 vision [12]. It describes the expected evolution of Intel processor architectures from single core systems, via multi-core systems toward many-core systems (see Figure 2.2). The Cell architecture [22] that is used in the PlayStation 3 is another example that shows the increasing popularity of MP-SoCs. It combines a PowerPC core with 8 synergetic processors that are used for data-intensive processing. A third example is the Nexperia digital video platform [14] from NXP. It supports digital television, home gateway and networking, and set-top box applications. An advanced member of the Nexperia family is the PNX8550 that combines two TriMedia processors, a MIPS processor and several hardware accelerators in a single chip.

The growing complexity of embedded multimedia systems leads to a large increase in their development effort. At the same time, the market dynamics for these systems push for shorter and shorter development times. It will soon be obligatory to keep to a strict design time budget that will be as small as six months to go from initial specification to a final and correct implementation [15]. Furthermore, the non-recurring engineering cost associated with the design and tooling of complex chips is growing rapidly. The International Technology Roadmap for Semiconductors (ITRS) predicts that while manufacturing complex Systems-on-Chip will be feasible, the production cost will grow rapidly as the costs of masks is raising drastically [16]. To address these issues,

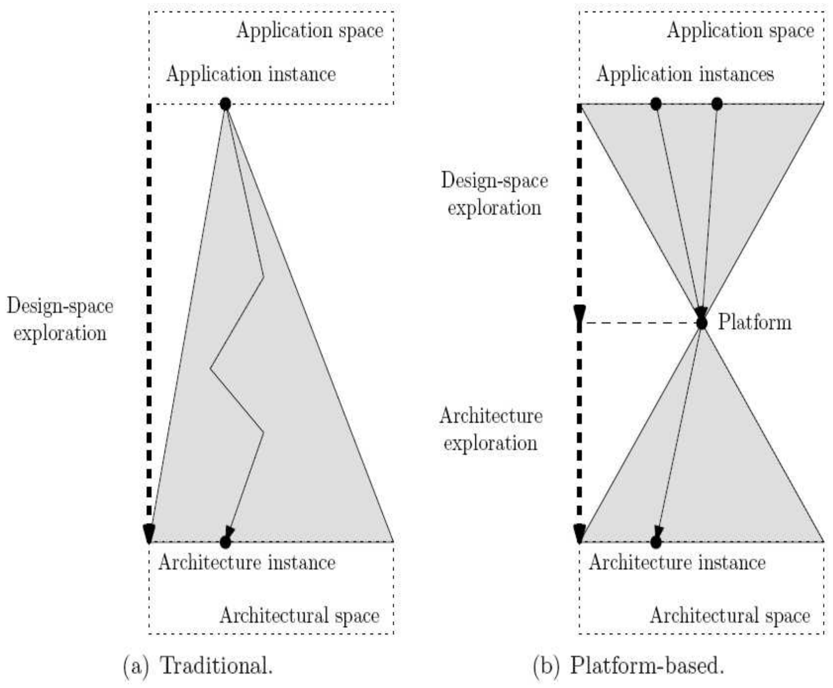


Figure 2.3: Design-space exploration strategies.

a platform- based design methodology is proposed in [17], [15]. The objective of this design methodology is to increase the re-use across different products that share certain functionality and the re-use between different product generations. The first form of re-use decreases the production cost as the same hardware can be used in more products. The second form of re-use lowers the development time as functionality implemented for a product does not have to be re-implemented for a successor product. The traditional design methodology is a single monolithic flow that maps an application onto an architecture (see Figure 2.3(a)). It starts with a single application which is shown at the top of Figure 2.3(a). The bottom of the figure shows the set of architectures that could support this application. The design process (black arrow) selects the most attractive solution as defined by a cost function. Synthesis of this architecture is often an explicit objective of the design methodology [7], [18].

The platform-based design methodology [17], [15] no longer maps a single application to an architecture that is optimal for this single application. Instead, it maps an application onto a hardware/software platform that can also be used for different applications from the same application space (see Figure 2.3(b)). This platform consists of a set of interconnected hardware components (e.g., processors, memories, etc.), potentially software components with, for example, operating-system type of functionality and an application program interface (API) that abstracts from the underlying hardware and software. This API allows replacing one platform instance from the architecture space with another platform instance without the need to re-implement the application on the platform. The platform-based design methodology stimulates the use of a common ”platform” denominator between multiple applications from the same application space. As a result, future design flows that map an application to a platform will focus on compiling an application onto an existing platform [15].

The trends signaled above show that the context in which applications are executed is becoming more dynamic. In future systems, multiple applications are running concurrently on an MP-SoC, and the set of active applications may change over time. At the same time, users expect a reliable behavior [5] of each individual application independent of the context in which it is operating. Virtualization of the resources in a system has been proposed as a concept to tackle this problem. The idea behind virtualization is that an application is given the illusion that it is running on its own copy of the real hardware which howeverhas only a fraction of the resources that are available in the real platform. For example, a processor which can do 100 million instructions per second could use a Time-Division Multiple-Access (TDMA) scheduler to present itself to an application A as a processor which can run 50 million instructions per second. This leaves room for another application B to use the remaining 50 million instructions per second without knowing that application A is also running on this processor. Virtualization has become popular in recent years in server and desktop computers [12], [21]. The concept is also employed in embedded systems. The Cell architecture [22] of IBM uses virtualization to avoid that programmers have to think about sharing processing resources and to guarantee the real-time response characteristics of applications. The Hijdra architecture [23] of NXP is another example of an embedded multi-processor system that uses virtualization. This architecture assumes that every hardware component has a scheduler that allows it to be shared between applications without them influencing each others timing behavior.

In summary, the following trends in the design of embedded systems are observed from a design perspective.

- Heterogeneous multi-processor systems are used to provide the required computational power for novel embedded multimedia systems.
- Networks-on-chip are used to provide a scalable interconnect with timing guarantees between the processors in the system.
- Platform-based design reduces production cost, design cost and design time of embedded systems.

- Virtualization of resources is used to guarantee a predictable behavior of applications in a dynamic environment.

Bibliography

[1] *In-Stat. Consumer electronics sales. http://www.itfacts.biz/ index.php?id=P8224.*

[2] H. De Man. System design challenges in the post-PC era. *37th Design Automation Conference*, DAC 00, Proceedings, page x. ACM, 2000.

[3] M. Weiser. Ubiquitous computing. *Computer*, 26(10):7172, October 1993.

[4] U. Hansmann and M. S. Nicklous and T. Stober. Pervasive computing handbook. *Springer-Verlag*, January 2001.

[5] E. Aarts and R. Harwig and M. Schuurmans. Ambient Intelligence. *The Invisible Future: The Seamless Integration of Technology in Everyday Life*, McGraw-Hill, 2002, pages 235250.

[6] T. Basten and M. C. W. Geilen and H. W. H. de Groot Ambient Intelligence: Impact on Embedded System Design. Kluwer Academic Publishers, November 2003.

[7] W. Thies amd M. Karczmarek and S. Amarasinghe Streamit: A language for streaming applications. *11th International Symposium on Compiler Construction*, CC 02, Proceedings, volume 2304 in LNCS, pages 179196. Springer-Verlag, 2002.

[8] F. Vaandrager. Lectures on Embedded Systems LNCS 1494, chapter Introduction, pages 13. Springer-Verlag, 1998.

[9] R. J. Bril and C. Hentschel and E. F. M. Steffens and M. Gabrani and G. van Loo and J. H. A. Gelissen. Multimedia QoS in consumer terminals. *In Workshop on Signal Processing Systems*, Proceedings, pages 332343. IEEE, 2001.

[10] T. Basten and L. Benini and A. Chandrakasan and M. Lindwer and J. Liu and R. Min and F. Zhao. Scaling into ambient intelligence. *In Conference on Design Automation and Test in Europe*, DATE 03, Proceedings, pages 7681. IEEE, 2003.

[11] A. A. Jerraya and W. Wolf Multiprocessor Systems-on-Chip. Elsevier, September 2005.

[12] S. Y. Borkar and H. Mulder and P. Dubey and S. S. Pawlowski and K. C. Kahn and J. R. Rattner and D. J. Kuck. Platform 2015: Intel processor and platfrom evolution for the next decade. Technical report, Intel, 2005.

[13] J. A. Kahle and M. N. Day and H. P. Hofstee anf C. R. Johns and T. R. Maeurer and D. Shippy. Introduction to the cell multiprocessor. IBM Journal of Research and Development, 49(4):589604, 2005.

[14] S. Dutta and R. Jensen and A. Rieckmann. Viper: A multiprocessor SoC for advanced set-top box and digital tv systems. IEEE Design and Test of Computers, 18(5):2131, September 2001.

[15] K. Keutzer and S. Malik and A. R. Newton and J. M. Rabaey and A. Sangiovanni-Vincentelli. System-level design: Orthogonalization of concerns and platform-based design. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 19(12):15231543, December 2000.

[16] ITRS. International technology roadmap for semiconductors 2005 edition. Technical report, ITRS, 2005.

[17] A. Ferrari and A. Sangiovanni-Vincentelli. System design: Traditional concepts and new paradigms. In International Conference on Computer Design, ICCD 99, Proceedings, pages 212. IEEE, 1999.

[18] R. Ernst and J. Henkel and Th. Benner and W. Ye and U. Holtmann and D. Herrmann and M. Trawny. The COSYMA environment for hardware/software cosynthesis of small embedded systems. Microprocessors and Microsystems, 20(3):159166, May 1996.

[19] F. Balarin and M. Chiodo and P. Giusto and H. Hsieh and A. Jurecska and L. Lavagno and C. Passerone and A. Sangiovanni-Vincentelli and E. Sentovich and K. Suzuki and B. Tabbara Hardware-Software Co-design of Embedded Systems: The POLIS Approach. Kluwer Academic Publishers, June 1997.

[20] A. Avizienis and J. -C. Laprie and B. Randell and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. Transactions on Dependable and Secure Computing, 1(1):1133, January 2004.

[21] J. L. Hennessy and D. A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 4th edition, 2007.

[22] J. A. Kahle and M. N. Day and H. P. Hofstee and C. R. Johns and T. R. Maeurer and D. Shippy. Introduction to the cell multiprocessor. IBM Journal of Research and Development, 49(4):589604, 2005.

[23] M. Bekooij and R. Hoes and O. Moreira and P. Poplavko and M. Pastrnak and B. Mesman and J. D. Mol and S. Stuijk and V. Gheorghita and J. van Meerbergen. Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices. Chapter Dataflow Analysis for Real-Time Embedded Multiprocessor System Design, pages 81108. Springer, May 2005.

Chapter 3

Application-Specific Power-Aware Workload Allocation for Voltage Scalable MPSoC Platforms

3.1 Overview

In this chapter, we address the problem of selecting the optimal number of processing cores and their operating voltage/frequency for a given workload, to minimize overall system power under application-dependent QoS constraints. Selecting the optimal system configuration is non-trivial, since it depends on task characteristics and system-level interaction effects among the cores. For this reason, our QoS-driven methodology for power aware partitioning and frequency selection is based on functional, cycle-accurate simulation on a virtual platform environment. The methodology, being application-specific, is demonstrated on the DES (Data Encryption System) algorithm, representative of a wider class of streaming applications with independent input data frames and regular workload.

3.2 Introduction

Many state-of-the-art or envisioned Multi-Processor Systems-on-Chip (MPSoCs) adopt the symmetric multi-processing paradigm [8]. This is due to the evolving micro-architecture of integrated cores, to the extension of their instruction set architecture in the direction of DSPs and to the increasing levels of integration made available by technology scaling. The design and implementation of MPSoCs is characterized by conflicting requirements of the ever increasing demand for higher performance and stringent power budgets. Circuit-level power minimization techniques can be used to address the power concern, including clock gating [18], dynamic voltage and frequency scaling (DVFS) [20] and low voltage design with variable/multiple V_{dd}/V_{th} control [16]. Furthermore, CMOS technology progressively allows an increasing number of voltage and clock domains to be specified on the same chip (see the *voltage islands* concept in [12]).

Lowering supply voltage reduces power quadratically but also results in a performance degradation, which translates into a reduction of the processor operating frequency at which functional correctness is guaranteed. Therefore, voltage scaling is usually associated with frequency scaling and vice versa.

In the new MPSoC domain, the problem of voltage and frequency selection cannot be optimally solved if we consider each processor in isolation. First, tasks running on individual cores are tightly related, since they are often the result of an application partitioning process, based on a specific workload allocation policy which creates task inter-dependencies.

Second, system-level interaction among the variable-voltage/frequency cores might induce non-trivial effects on global system performance and energy metrics. As an example, system performance is a non-additive metric, but strongly depends on the inter-processor synchronization mechanism and on the interaction on the system bus of the traffic patterns generated by cores running at different speeds.

However, in a parallel computing domain like MPSoCs, workload allocation is another degree of freedom for system power minimization. DFVS and extraction of task level parallelism should be jointly addressed in a global power minimization framework. Here, the trade-off to span is between the number of concurrent processors and the power overhead they introduce in the system, which is a function of their clock speed. For instance, the same application-dependent throughput constraint could be met by means of N processors working at speed X or by sharing the workload among $N + M$ concurrent processors working at reduced speed $X^{'}$.

In this chapter, we take a semi-static approach to the frequency/voltage selection problem. Pareto-optimal processor configurations are statically determined at design time in the power-performance exploration space, and for different bus traffic conditions. However, our design time exploration framework opens the way for a dynamic update of frequency settings as a function of the varying features of interfering traffic, switching to the statically derived

Pareto-optimal configuration for the new working conditions.

This approach can be applied to application domains characterized by regular and highly predictable workloads, with minimum run-time fluctuations. Baseband processing in wireless modems, encryption engines, digital image filtering and many signal processing functions are examples thereof. For such applications, performance fluctuations can be only induced by interfering events, such as additional data transfers generated on the bus by other running applications. In fact, although we consider a maximum number of available processors, we assume that not all of them must be necessarily allocated to the execution of a given application. We introduce a complete QoS-based methodology that provides the optimal number of processing cores for a given scalable workload and their individual frequency/voltage settings in such a way to minimize system power while meeting application throughput constraints.

Our methodology for processor allocation and frequency/voltage selection is simulation based. We want to overcome the limitations of previous works, which proposed theoretical and highly abstract models without validation on real platforms or on functional, timing accurate MPSoC simulation tools. In contrast, we deployed a virtual platform [26], enhanced with hardware extensions for variable frequency/voltage cores, for developing the allocation and frequency selection methodology and for validating our approach. As such, our methodology is strongly related to the specific workload. We therefore restricted our analysis to the optimization of a parallel, highly scalable DES encryption algorithm, deriving a methodology and drawing conclusions that can be extended to the whole class of applications DES algorithm belongs to, namely streaming applications with uncorrelated input data frames.

This chapter is structured as follows. Section 2 reports previous work while the virtual platform environment is described in Section 3. Section 4 and 5 present DES algorithm and problem formulation. Section 6 explains our methodology, whose results are reported in Section 7.

3.3 Related Work

A survey of techniques for system level power optimization is reported in [1, 2]. The issue of voltage/frequency selection for single-processor systems is a mature research field: many run-time dynamic techniques have been proposed [3, 4], and validation tools [5] and hardware [32] are available.

On the contrary, in the multi-processor system domain, many approaches based on theoretical analysis and abstract simulation have been proposed, but an accurate validation of the effectiveness of these techniques is still in its early stage.

In [6] the problem of minimizing power of a multi-processor system using multiple variable supply voltages is modelled as a mixed integer non-linear programming optimization problem. The work in [7] points out that minimizing communication power without considering computation may actually lead to higher energy consumption at the system level. An analytical approach is taken in [8] to assign single optimal voltage to each processor present in an application-specific heterogeneous multi-processor system after allocation and scheduling have been performed. A heuristic to address the problem of energy-efficient voltage scheduling of a hard real-time task graph with precedence constraints for a multi-processor environment is presented in [9], but it is limited to dual voltage systems. The algorithm introduced in [11] targets power utilization and performance of multi-processor systems wherein parameters such as operating voltage, frequency and number of processors can be tuned. Approaches for combined DVS and adaptive body biasing in distributed time-constrained systems have been reported in [29]. A technique for combined voltage scaling of processors and communication links, taking into account dynamic as well as leakage power consumption, is proposed in [30]. The energy-aware scheduling algorithm presented in [17] consists of a design-time phase, which results in a set of Pareto-optimal solutions, and of a run-time phase, that uses them to find a reasonable cycle budget distribution for all of the running threads. The effect of discrete voltage/speed levels on the energy savings for multi-processor systems is investigated in [14], and a new scheme of slack reservation to incorporate voltage/speed adjustment overhead in the scheduling algorithm is also proposed. The approach to energy minimization in variable voltage MPSoCs taken in [13] consists of a two phase framework that integrates task assignment, ordering and voltage selection. Aydin et al. [23] propose energy-efficient periodic real-time task scheduling algorithms based on the earliest deadline first scheduling strategy. Heuristic algorithms are instead used in [25]. [17] proposes to schedule real time tasks with precedence constraints by means of list heuristics. Many works assume that actual execution time of tasks to be scheduled is equal to the worst case execution time [25, 26, 28]. A more realistic approach is taken in [24].

With respect to previous work, our contributions are: 1) the joint solution of processor allocation and frequency/voltage setting; 2) a novel algorithm for efficient construction of the Pareto frontier for selecting the optimal system operating points based on throughput and utilization constraints; 3) validation on a full-system functional and power simulation for a real application case study.

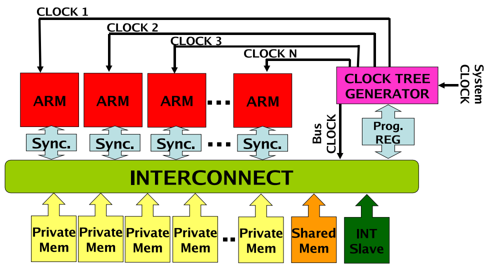


Figure 3.1: MPSoC platform with hardware support for frequency scaling.

3.4 Virtual Platform Environment

We carried out our analysis within the framework of the SystemC-based MPARM simulation platform [26]. Figure 3.1 shows a pictorial overview of the simulated architecture. It consists of a configurable number of 32-bit ARMv7 processors. Each processor core has its own private memory, and a shared memory is used for inter-processor communication. Synchronization among the cores is provided by hardware semaphores implementing the *test-and-set* operation. The system interconnect is a shared bus instance of the STBus interconnect from STMicroelectronics. The software architecture consists of an embedded real-time operating system called RTEMS [33], which natively supports synchronization and inter-task communication primitives.

The virtual platform environment provides power statistics leveraging technology-homogeneous power models made available by STMicroelectronics for a 0.13 μ m technology for ARM cores, caches, on-chip memories and the STBus.

Support for Variable Frequency Cores. The virtual platform has been extended to support different working frequencies for each processor core. For this purpose, additional modules were integrated into the platform, namely a variable clock tree generator, programmable registers and a synchronization module.

The clock tree generator feeds the hardware modules of the platform (processors, buses, memories, etc.) with independent and frequency scaled clock trees. The frequency scaled clock trees are generated by means of frequency dividers (shift counters), whose delay can be configured by users at design-time. A set of programmable registers has been connected to the system bus to let the operating system or a dedicated hardware module (monitoring system status) select the working frequencies. Each one of these registers contains the integer divider of the baseline frequency for each processor.

Scaling the clock frequency of the processors creates a synchronization issue with the system bus, which is assumed to work at the maximum frequency. The processing cores and the bus interface communicate by means of a handshaking protocol which assumes the same working frequency at both sides. Therefore, a synchronization module was designed, containing two dual-ported FIFOs wherein data and addresses sent by the bus interface to the processor and vice versa are stored. This module works with a dual clock: one feeding the core side and one feeding the bus interface side. Finally, the module also takes care of properly interfacing processor to bus signals, and a dedicated sub-module is implemented for this purpose. In Figure 3.1, the hardware extensions for frequency-scaled cores have been shaded. The Maximum STBus frequency of 200MHz was kept as the maximum processing core frequency, to which frequency dividers were applied. The scaling factor for the power supply was derived from [19].

3.5 Workload Allocation

3.5.1 DES Dataflow

DES performs two main operations on input data, controlled by the key: bit shifting and bit substitution. By doing these operations repeatedly and in a non-linear way, the final result cannot be used to retrieve the original data without the key. DES works on chunks of 64 bits of data at a time. Each 64 bits of data is iterated on from 1 to 16 times (16 is the DES standard). For each iteration, a subset of the key is fed into the encryption block, that performs several different transforms and non-linear substitutions. More details can be found in [31].

3.5.2 Mapping DES on MPSoC Platform

DES algorithm matches the master-slave workload allocation policy. DES encrypts and decrypts data using a 64-bit key. It splits input data into 64-bit chunks and outputs a stream of 64-bit ciphered blocks. Since each input element (called *frame*) is independently encrypted from all others, the algorithm can be easily mapped onto a variable number of cores. Moreover, DES poses a balanced workload to the processors, since the execution time is almost independent on input data. This consideration will be important when we will address core clock scaling later in this chapter.

In the parallelized version of DES, we define three kinds of tasks. An initiator task (*producer*) dispatches 64-bit blocks together with a 64-bit key to n calculator tasks (referred as *working* tasks) for encryption. Initiator task and working tasks are allocated to different processing elements (PEs). A buffer in shared memory is used to exchange data. Since frames are uncorrelated from each other, computation of working tasks can be carried out in parallel by running multiple task instances on different slave processors. Here slave processors just need to be independently synchronized with the producer, which alternatively provides input data to all of the slave tasks. Finally, a collector tasks (*consumer*) reconstructs an output stream by concatenating the ciphered blocks provided by the working tasks. It is allocated onto another PE and communicates with workers by means of output buffers.

Both input and output buffers are located in shared memory. Each one of them is implemented using two queues, so that one queue can be write-accessed by the producer while the other one is read-locked by the worker. The same holds for the output buffer. Moreover, in case of multiple workers, each one has its own input and output buffer. In brief, the streaming application is mapped onto the platform as a three-stage pipeline, where the intermediate stage can be made by an arbitrary number of multiple parallel tasks. The overall system model is described in Figure 3.2.

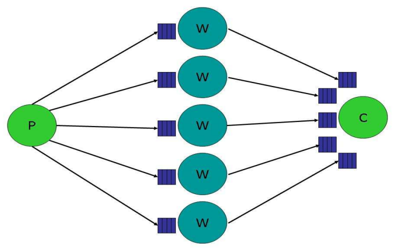


Figure 3.2: DES workload allocation policy.

3.6 Problem Formulation

The ultimate objective of this work is to find the optimal number of parallel workers (and a corresponding number of slave processors) to achieve a given throughput (in frames/sec), and to set the operating frequencies of the cores in the system (producer, consumer and workers) so to minimize overall power consumption. We decided to make producer and consumer work at the same speed, in order to keep the system stable. Moreover, since the workload of the parallel workers is balanced due to the intrinsic DES characteristics, we assume they are also running at the same speed.

The optimization problem can be thus formulated as follows. For a given throughput (T), which univocally determines consumer speed (and producer), we need to find a couple (N_{WK}, S), where N_{WK} is the number of workers and S is the scaling factor between producer (or consumer) speed and workers' speed. If f_{WK} is the clock frequency of workers and f_{PR} is the clock frequency of producer/consumer, we obtain $f_{WK} = S \cdot f_{PR/CN}$.

Given this formulation, the optimization problem can be solved by searching for those system configurations that minimize the following cost function:

$$P = F(N_{WK}, f_{WK}, f_{PR}, f_{CN}), \tag{3.1}$$

where P is average power consumption of the whole MPSoC, and the following constraints hold:

$$f_{PR} = f_{CN} = f_{WK}/S; \tag{3.2}$$

$$N_{WK} \leq N_{MAX}; \tag{3.3}$$

where N_{MAX} is the maximum number of PEs available in the system, excluding the PEs allocated to producer and consumer tasks. In the following, we will change this constraint to handle the case in which not all system PEs can be reserved for DES application.

The intuition behind our approach is that configurations that minimize the cost function are those that minimize system idleness. As a consequence, the optimum scaling factor S, that is a function of N_{WK}, will be the one that best synchronizes the execution of workers with that of producer/consumer. In general, since the computational effort required by each worker to produce an output frame is much larger than that required by the producer and consumer, S will be larger than one for low N_{WK} and lower than one for high N_{WK}. This is because producer and consumer tasks are essentially memory-bound, while working tasks are CPU-bound. For a memory-bound task, throughput is less sensitive to frequency scaling, since this latter does not affect memory access times, that stay constant. Once S has been found, the absolute speed values are determined by the required throughput.

As showed in the experimental result section, the solution to this problem is not trivial. Simple solutions that tune either N_{WK} or core frequencies in isolation are sub-optimal.

Handling Bus Effects. In general, we cannot assume that all system resources are available for the target application. This fact has a double effect on the previously defined optimization problem. First of all, not all of the PEs are available, so that equation 3.3 changes as follows:

$$N_{WK} \leq N_{FREE}; \tag{3.4}$$

where N_{FREE} is the number of free PEs. Note that our methodology will indicate whether it is power efficient to use all of the N_{FREE} PEs or a lower number of them, together with the proper frequency settings, for a given workload. Since the bus is partially occupied by interfering traffic generated by applications running on the busy PEs, this will affect the performance of the communication among DES tasks. As a results, the cost function depends on traffic conditions. We can characterize bus traffic by means of two parameters: ρ is the bus bandwidth consumed by interfering traffic, while σ is the average burst size of the interfering traffic. We can then rebuild Equation 3.5 as follows:

$$P = F(N_{WK}, f_{WK}, f_{PR}, f_{CN}, \sigma, \rho). \tag{3.5}$$

3.7 QoS-Driven Optimization Strategy

Our power optimization framework consists of a two step process. First, we statically perform a smart exploration to find Pareto-optimal configurations that minimize the cost function in a power/throughput design space. We perform this exploration by varying traffic parameters (σ, ρ) in a discrete range of possible values. Then, we store these configurations in a three-dimensional look-up table indexed by σ, ρ and N_{FREE} that will be used at run-time in a semi-static way to maintain QoS under varied traffic conditions.

3.7.1 Design-time Smart Design Space Exploration

We explore the throughput/power design space in an attempt to find the Pareto curve. Each exploration assumes a given number of workers, therefore we have to perform multiple exploration rounds corresponding to a different number of available workers in the system. Our method tries to find the optimal scaling factor between producer/consumer and workers frequency that gives rise to minimum idleness of processing cores during system operation. For each explored configuration in the design space, a simulation run is performed to compute the corresponding level of power dissipation of the overall system. However, as will be explained shortly hereafter, our approach cuts down on the number of configurations to analyze in order to determine the Pareto points.

The methodology starts by simulating the configuration where all of the PEs (both workers and producer/consumer) run at the maximum speed, then derives two other configurations by scaling PR/CN or WK frequency. From

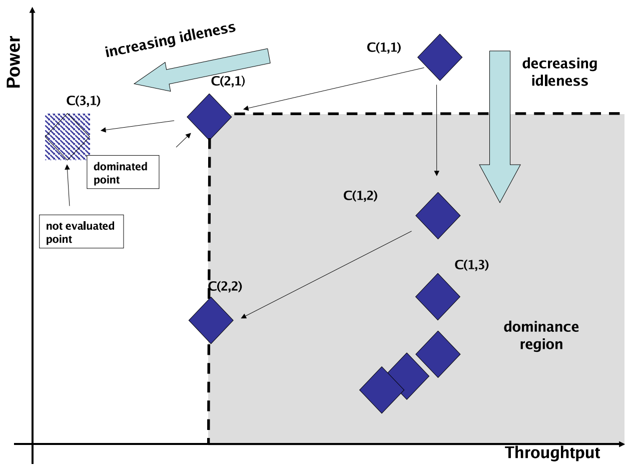


Figure 3.3: Example of smart exploration.

these new configurations, we generate other configurations using the same scaling rule. Clearly, since we scale the frequency of PEs, at each step we move to lower throughput regions. However, in order to reduce the number of simulation runs, we defined an optimized version of the algorithm that allows to reduce the number of design points to be explored.

The smart exploration is based on the following intuitions. First, a configuration is said to be ”dominated” if there is at least another configuration that provides a higher or equal throughput with lower power. Configurations that turn out to be non-dominated once the smart search procedure completes are those belonging to the Pareto curve. Second, it is worth observing that dominated configurations cannot generate (using the scaling rule explained above) non-dominated configurations that cannot be obtained by scaling non-dominated configurations already found. Hence, when we generate two new configurations, we always check if one of them is dominated. If this is true, the dominated configuration is discarded and thus all other configurations derived from it are not explored. Should some of the unexplored points belong to the Pareto curve, we will find them by scaling the non-dominated configuration.

The observation can be justified as follows (refer to Figure 3.3). Let us first point out that derived configurations can have lower or equal throughput w.r.t. their parent configurations, being obtained through frequency scaling. Then, let us indicate with (i, j) the scaling factor of PR/CN and WK respectively with respect to maximum system frequency. Assume now to generate two new configurations C(1,2) and C(2,1) from the starting one C(1,1). If a configuration is dominated, for example C(2,1), this means that its power consumption is higher and the throughput is lower than another one already explored, namely C(1,2). In practice, C(2,1) corresponds to a less efficient operating point in which the amount of power wasted in idle cycles is increased w.r.t. C(1,1). Since in C(2,1) we have scaled the PR/CN frequency, we deduce that the performance bottleneck in C(1,1) was represented by the producer/consumer, and that by moving to C(2,1) we have made the pipeline even more unbalanced. In contrast, C(1,2) reduces workers frequency, and goes in the direction of balancing the pipeline and minimizing idleness. Suppose now to generate two additional configurations from the dominated point C(2,1). Since we are further decreasing PR/CN frequency, we are moving away from the optimal ratio between PR/CN and WK speed, increasing the power wasted by WKs in idle cycles.

These considerations are at the core of the smart exploration algorithm described in Figure 3.4. It makes use of two lists of configuration points. A ″Pareto list″ stores configurations that pass all dominance checks and are therefore proved to be Pareto points. A ″temporary list″ instead stores candidate Pareto points, which might turn out to be dominated by points that still have to be generated by the frequency scaling methodology. When we are sure that the scaling process will not generate configurations providing higher throughput than that ensured by a configuration in the temporary list, then this latter can be moved to the Pareto list.

A generic loop of this algorithm works as follows. From each configuration C(i,j) we generate two new configurations C(i+1,j) and C(i,j+1), and denote the one with highest throughput as *current configuration* (C_{CURR}). The other one is stored in the temporary list (sorted for decreasing throughput) only if it passes a dominance check against C_{CURR}, against all previously stored configurations in the temporary list and the ones that are already in the Pareto list. Even though it passes the dominance check, this point cannot be put in the Pareto list since there could be other dominating configurations with intermediate throughput values generated by C_{CURR}.

Before examining C_{CURR} we must make sure that there are no configurations in the temporary list with higher throughput. If this is the case, such configurations should be examined first (and eventually added to the Pareto list), so that when we later perform the dominance check on C_{CURR} and it turns out to be successful, C_{CURR} can be directly inserted in the Pareto list. If two configurations in the Pareto list have the same throughput, the one with the lowest power replaces the other one. New configurations are generated by scaling the last one added to the Pareto list.

At the end of this process, we obtain a Pareto curve, and the procedure has to be repeated for a different number of workers, exploring all cases for which $N_{WK} \leq N_{FREE}$. By composing all these Pareto curves, we observe that in each

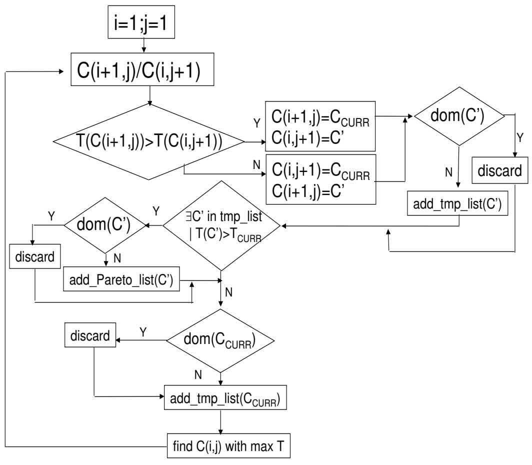


Figure 3.4: Flow diagram of the smart exploration algorithm.

region of the configuration space some Pareto curves happen to stay above the other ones. For instance, when small throughput values have to be ensured, employing an overly high number of workers is power-inefficient, since the same performance can be provided by fewer workers. In other words, the composition of all curves leads to an overall Pareto curve (PAR^O) showing the optimal number of workers and the optimal operating frequencies of the cores that allow to provide a given throughput at minimum power.

The reduction on exploration time allowed by our algorithm can be exploited to perform more exploration rounds accounting for the impact on the power-performance trade-off of system parameters that are likely to dynamically change at run-time. In particular, we decided to derive multiple overall Pareto curves PAR^O in presence of different traffic patterns (σ, ρ), thus allowing semi-static resource allocation and frequency setting based on the levels of bus traffic. Obviously, the discretization of explored traffic patterns (whose entity is related to the size of look-up tables) is unavoidable.

3.7.2 QoS-Oriented Semi-Static Workload Allocation

Once Pareto-optimal configurations have been statically determined in the previous step, they will be stored in a three-dimensional look-up table having traffic parameters and N_{FREE} as indexes. The table returns the overall Pareto curve for N_{FREE} maximum processors, from which the optimal configuration for a given throughput constraint can be immediately derived.

At run-time, working conditions might change as an effect of events occurring at a large time granularity, such as freed PEs or newly admitted appli-

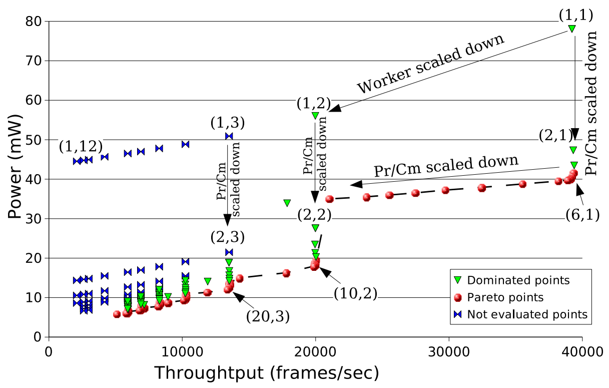


Figure 3.5: Pareto curve with one worker.

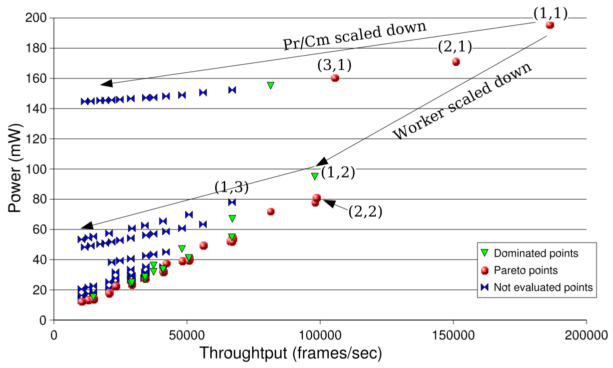


Figure 3.6: Pareto curve with five workers.

cations in the system or abrupt changes in bus traffic due to data-dependent applications. In this case, the amount of resources allocated for our application and their frequency settings must be recomputed by looking at the table. Traffic parameters and number of free cores might be retrieved from the environment (i.e. an operating system) or through dedicated monitoring hardware. Since we do not store in the table all possible values of σ, ρ but only a discrete set, it is possible that run-time values of traffic parameters do not belong to this set. In this case, Pareto-optimal points must be obtained either by interpolating among stored configurations or by selecting conservative configurations that provide the target throughput under traffic conditions that are worst than those actually detected. This latter approach of course incurs power penalties that are the price to pay for the discretization of stored values.

3.8 Results

In this section we first show power/performance Pareto curves obtained through smart design space exploration that are used to fill in the look-up tables described in previous section. We consider power contributions of all system components. Then, we compare our results with those provided by alternative approaches. As mentioned in Section 3.4, our platform uses frequency dividers to generated scaled clocks for the cores, as done in common embedded platforms [22]. Scaling is obtained by dividing maximum clock speed by an integer number, so that speed levels are discretized. Such a discretization will play a significant role in determining the Pareto curves.

3.8.1 Pareto Optimal Configurations

Pareto-optimal configurations are shown in Figure 3.5 for one worker (PAR^1) and in Figure 3.6 for 5 workers (PAR^5). Both figures outline the effectiveness of the smart design space exploration process. It is evident that a large number of configurations have not been evaluated, thus cutting down on simulation runs. Points that have been discarded because they are dominated are also showed. Let us focus on the one worker case in Figure 3.5. The algorithm starts from the upper rightmost point corresponding to $f_{PR/CN}$ = f_{WK} = f_{MAX}. It immediately discards the upper points in the plot and moves down vertically until it finds the first point of the Pareto curve. We observed that the discarded points correspond to f_{WK} scaling. This means that in the starting configuration the single worker is the bottleneck and by scaling down f_{WK} we make the system even more unbalanced. In contrast, by scaling $f_{PR/CN}$, we get a reduction of power consumption with constant throughput until PR and CN become the bottlenecks. This corresponds to a Pareto optimal configuration, and the relative scaling factor between $f_{PR/CN}$ and f_{WK} frequencies minimizes the idleness. All configurations with same throughput but higher power are then discarded. Other points that we obtain by scaling PR/CN from here increase the idleness, but they are still Pareto optimal since they are not dominated by other configurations. The reason for this is the discrete number of available frequencies.

With a larger number of workers (greater than four), Pareto curves become similar to Figure 3.6, where a case with 5 workers is represented. Here, PR/CN are the bottlenecks in the starting configuration. Although we correctly scale workers frequency to balance the system, the scaling granularity is so coarse (scaling of 5 workers at a time) that after one scaling step the workers become the bottleneck. Therefore the identification of Pareto points here is much less intuitive than in Figure 3.5.

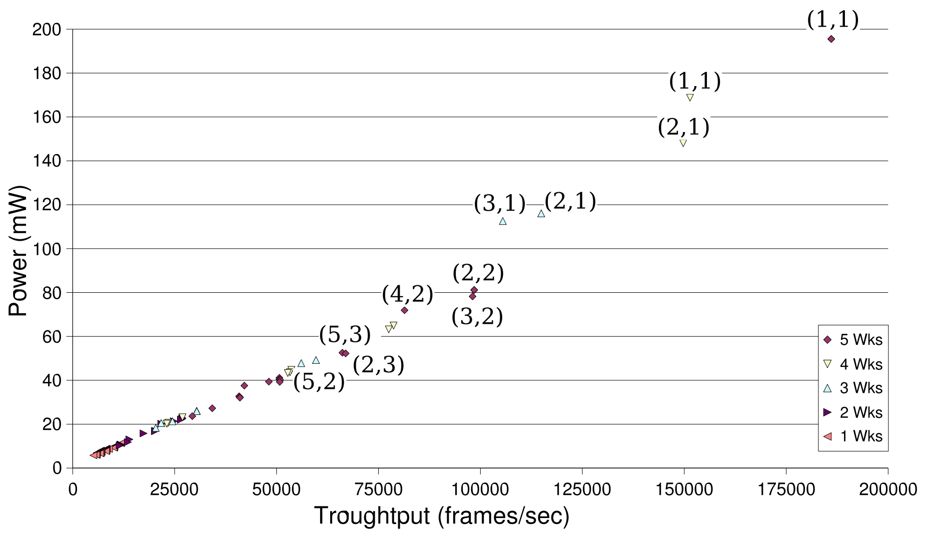


Figure 3.7: Overall Pareto curve with five available workers.

The overall Pareto curve ($PAR^O(N_{FREE})$) for a maximum number of available workers is obtained by comparing the Pareto-optimal configurations for the different numbers of workers. The curve is shown in Figure 3.7 for $N_{FREE} = 5$ workers. We can observe that using all workers is not always the most power-efficient solution.

Our analysis also shows that for a larger number of workers, second-order effects come into play. We in fact observed an increase of the achievable throughput until the number of workers is equal to eight, which is the last point with no diminishing returns. In fact, with more than eight WKs the bus saturates when high throughputs have to be delivered, as detailed in Figure 3.8. As a consequence, the configuration with 8 workers is the one which provides the highest throughput. By further increasing the number of workers, since the upper bound on the achievable throughput has been passed, the actual delivered throughput will not further increase. Moreover the power consumption will increase due to the contribution of additional cores. For this reason, when considering the overall Pareto curve $PAR^O(N_{MAX})$ with $N_{FREE} \geq 9$, configurations that do not use all of the workers are more efficient for higher througputs.

Interfering traffic effects on the power-performance trade-off points are shown in Figures 3.9 and 3.10. All the points correspond to configurations where producer, consumer and workers work at the same speed, which is the maximum speed on the rightmost part of the plot and a scaled speed as we move to lower throughput values. Figure 3.9 highlights the impact of reduced available bus bandwidth on DES performance for the 1 worker case. The throughput theo-

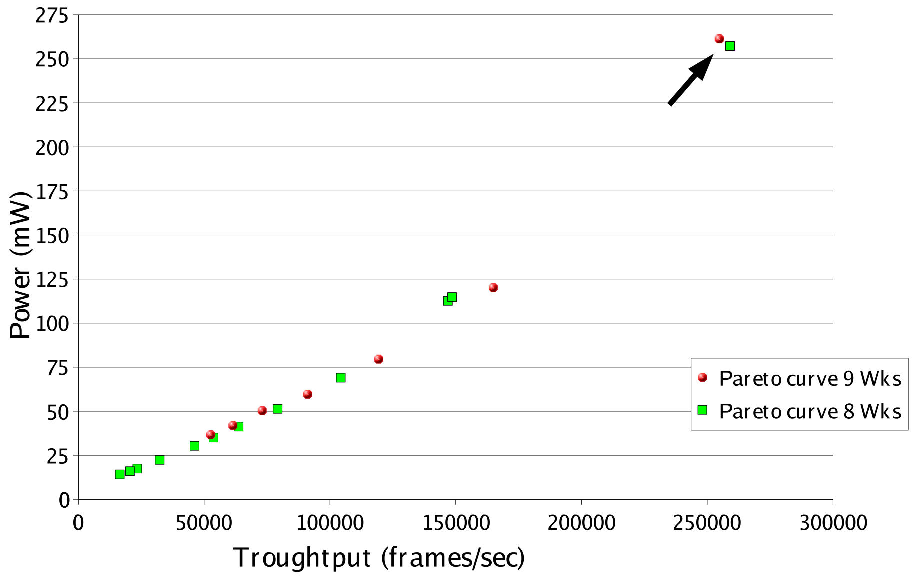


Figure 3.8: Bus saturation effect.

retically achievable by configurations on the rightmost part cannot be actually provided because of a larger impact of bus contention. This effect is not observed in the leftmost part, since there is a lower frequency of bus accesses to deliver lower throughput and since the processors are working at a lower speed than the bus.

Figure 3.10 instead shows the impact of average burst size of the interfering traffic (σ), keeping the bandwidth consumed by interfering traffic ρ constant. We show that the impact on throughput is larger for smaller but more frequent bursts, and that configurations providing high throughput values are more sensitive.

3.8.2 Efficiency Comparison

In Figure 3.11 we compare our optimal solutions for a 2 workers case ($N_{FREE} = 2$) with an alternative policy which always uses the maximum number of available workers and scales down all processor frequencies to get a lower throughput. With our power-aware methodology, we can cut down power by 30% for high throughput values, since we are able to reduce idleness. For lower throughput values the savings are smaller but still our configurations are more power efficient.

The effectiveness of the power-aware allocation has also been compared with a policy with no voltage/frequency scaling. In this case, a lower throughput can be achieved by employing a lower number of processors. Results are

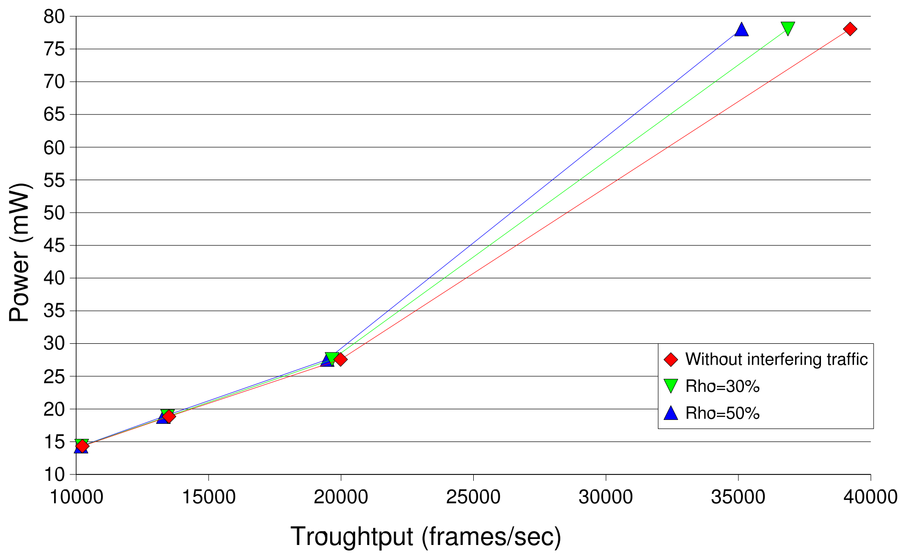


Figure 3.9: Effect of interfering traffic bandwidth. ρ is the bandwidth occupancy of interfering traffic with respect to maximum bus bandwidth.

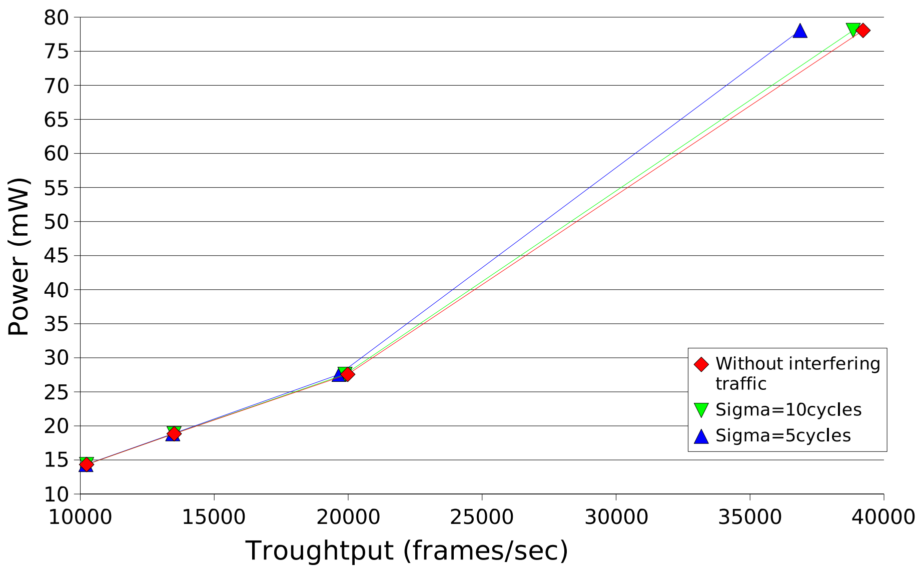


Figure 3.10: Effect of burst size of interfering traffic.

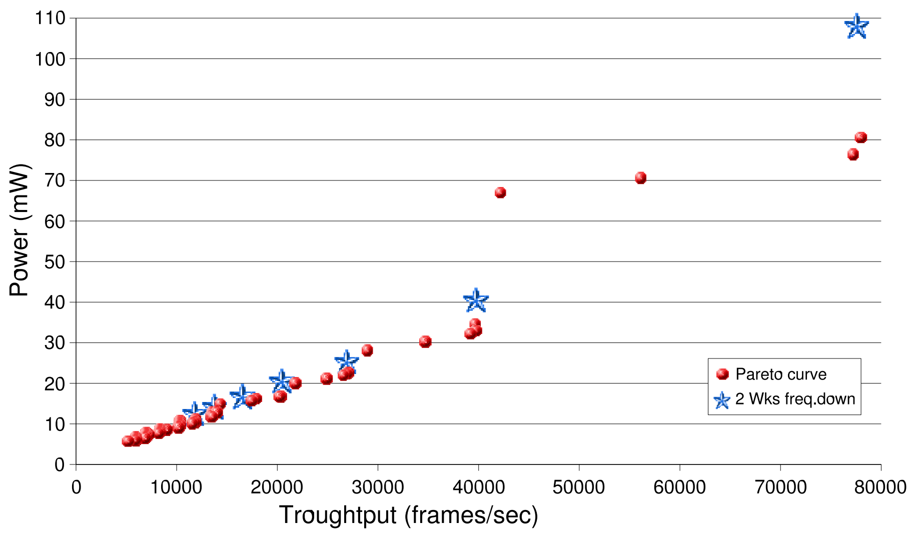


Figure 3.11: Comparison of power-aware allocation strategy with no workers tuning policy.

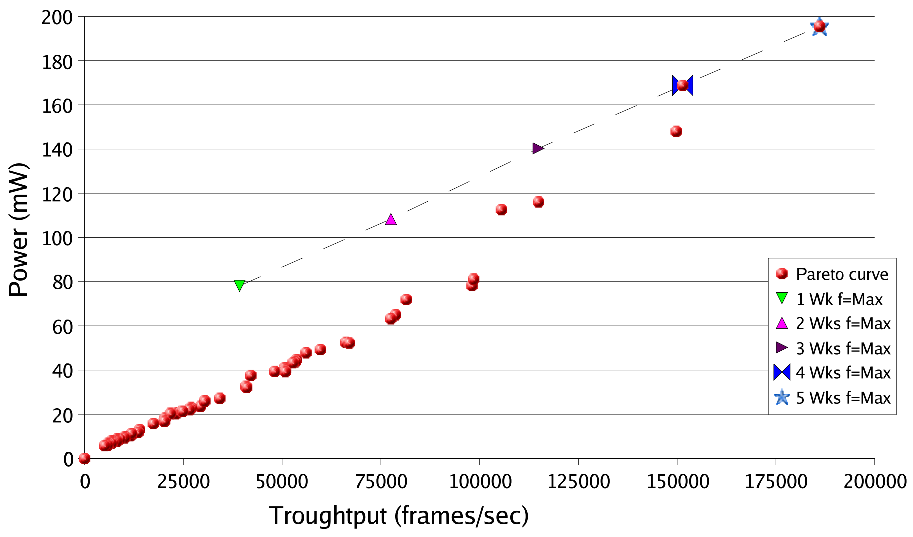


Figure 3.12: Comparison of power-aware allocation strategy with no frequency scaling policy.

reported in Figure 3.12 for $N_{FREE} = 5$. The comparison highlights that using our strategy allows to save 50% of power for lower throughput values, since we again scale frequencies to reduce idleness.

3.9 Conclusion

We presented a QoS-driven methodology for optimal allocation and frequency selection. Our methodology is based on functional simulation and full system power estimation. It is demonstrated on the DES algorithm, representative of a wider class of streaming applications with independent input data frames and regular workloads. We have showed the savings in terms of needed simulation runs and the efficiency with respect to alternative approaches.

Bibliography

[1] L. Benini, A. Bogliolo, and G. De Micheli. "A survey of design techniques for system-level dynamic power management". *IEEE Trans. on VLSI Systems*, pages 299–316, June 2000.

[2] N. Jha. "Low power system scheduling and synthesis". *IEEE/ACM Conf. on CAD*, pages 259–263, 2001.

[3] W. Kwon and T. Kim. "Optimal voltage allocation techniques for dynamically variable voltage processors". *IEEE Trans. on VLSI Systems*, pages 125–130, June 2003.

[4] P. Pillai and K. Shin. "Real-time dynamic voltage scaling for low-power embedded operating systems". *ACM SIGOPS 01*, pages 89–102, October 2001.

[5] A. Iyer and D. Marculescu. "Power efficiency of voltage scaling in multiple clocks, multiple voltage cores". *Int. Symposium of Computer Architecture*, pages 158–168, May 2002.

[6] L. Leung, C. Tsui, and W. Ki. "Simultaneous task allocation, scheduling and voltage assignment for multiple-processors-core systems using mixed integer nonlinear programming". *ISCAS03*, V:309–312, May 2003.

[7] J. Liu, P. Chou, and N. Bagherzadeh. "Communication speed selection for embedded systems with networked voltage-scalable processors". *CODES02*, pages 169–174, May 2002.

[8] A. Rae and S. Parameswaran. "Voltage reduction of application-specific heterogeneous multiprocessor systems for power minimisation". *ASP-DAC*, pages 147–152, January 2000.

[9] D. Roychowdhury, I. Koren, C. Krishna, and L. Y.H. "A voltage scheduling heuristic for real-time task graphs". *Int. Conf. on Dependable Systems and Networks*, pages 741–750, June 2003.

[10] D. Bertozzi and L. Benini. “Battery lifetime optimization for energy-aware circuits ”. *Low Power Electronics Design*, edited by C.Piguet, pages –, 2004.

[11] J. Suh, D. Kang, and S. Crago. “Dynamic power management of multi-processor systems”. *Int. Parallel and Distributed Processing Symp.*, pages 97–104, April 2002.

[12] D. Lackey, P. Zuchowski, D. Bedhar, T.R. aand Stout, S. Gould, and J. Cohn. “Managing power and performance for systems-on-chip designs using voltage islands ”. *Int. Conf. on CAD*, pages 195–202, November 2002.

[13] Y. Zhang, X. S. Hu, and D. Z. Chen. Task scheduling and voltage selection for energy minimization. *DAC03*, pages 183–187, 2003.

[14] D. Zhu, R. Melhem, and B. Childers. “Scheduling with dynamic voltage/speed adjustment using slack reclamation in multi-processor real-time systems”. *IEEE Trans. on Parallel and Distributed Systems*, 14:686–700, July 2003.

[15] D. Pham et al. “The design and implementation of a first generation CELL processor”. *IEEE/ACM ISSCC*, pp.184–186, 2005. July 2003.

[16] I. Hyunsik, T. Inukai, H. Gomyo, T. Hiramoto, and T. Sakurai. “VTC-MOS characteristics and its optimum conditions predicted by a compact analytical model”. *ISLPED01*, pages 123–128, August 2001.

[17] P. Yang, C. Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, and R. Lauwereins. “Energy-aware runtime scheduling for embedded-multiprocessor SOCs”. *IEEE Design and Test of Computers*, pages 46–58, 2001.

[18] L. Benini, P. Siegel, and G. De Micheli. “Automatic synthesis of gated clocks for power reduction in sequential circuits”. *IEEE Design and Test of Computers*, pages 32–40, December 1994.

[19] K. J. Nowka et al. “A 32-bit PowerPC System-on-a-Chip with support for dynamic voltage scaling and dynamic frequency scaling”. *IEEE JSSC* Vol. 37, no. 11, pp. 1441–1447, Nov. 2002.

[20] G. Qu. “What is the limit of energy saving by dynamic voltage scaling? ”. *IEEE/ACM Int. Conf. on Computer Aided Design*, pages 560–563, 2001.

[21] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, R. Zafalon, Analyzing On-Chip Communication in a MPSoC Environment, Proceedings of the Design, Automation and Test in Europe Conference and Exhibition 2004, Paris, France, Feb 16-20, 2004, pp. 752-757 Vol. 2

[22] i.MX21 Application Processor, http://www.freescale.com.

[23] H. Aydin, R. Melhem, D. Mosse', and P. Mejia-Alvarez. "Determining Optimal Processor Speeds for Periodic Real-Time Tasks with Different Power Characteristics ". *IEEE EuroMicro Conf. on Real-Time Systems*, pages 225, 2001.

[24] H. Aydin, R. Melhem, D. Mosse', and P. Mejia-Alvarez. "Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems ". *IEEE Real-Time Systems Symposium*, pages 95–105, 2001.

[25] H. Aydin, and Q. Yang. "Energy-Aware Partitioning for Multiprocessor Real-Time Systems ". *Int. Parallel and Distributed Processing Symposium*, pages 113–121, 2003.

[26] J.J. Chen, H.R. Hsu, K.H. Chuang, C.L. Yang, A.C. Pang, and T.W. Kuo. "Multiprocessor Energy-Efficient Scheduling with Task Migration Considerations". *EuroMicro Conf. on Real-Time Systems*, pages 101–108, 2004.

[27] F. Gruian, and K. Kuchcinski. "Lenes: Task Scheduling for Low Energy Systems Using Variable Supply Voltage Processors". *Asia South Pacific Design Automation Conf.*, pages 449–455, 2001.

[28] C.Y. Yang, J.J. Chen, and T.W. Kuo. "An Approximation Algorithm for Energy-Efficient Scheduling on a Chip Multiprocessor". *DATE05*, pages 468–473, 2005.

[29] A. Andrei, M. Schmitz, P.Eles, Z.Peng, and B.M. Al-Hashimi. "Overhead-Conscious Voltage Selection for Dynamic and Leakage Energy Reduction of Time-Constrained Systems". *DATE04*, pages 518–523, 2004.

[30] A. Andrei, M. Schmitz, P.Eles, Z.Peng, and B.M. Al-Hashimi. "Simultaneous Communication and Processor Voltage Scaling for Dynamic and Leakage Energy Reduction in Time-Constrained Systems". *ICCAD04*, pages 362–369, 2004.

[31] Federal Information Processing Standards Publication 46-2, "'Announcing the Standard for DATA ENCRYPTION STANDARD (DES)"', `http://www.itl.nist.gov/fipspubs/fip46-2.htm`, Dec. 1993.

[32] Intel XScale technology, http://www.intel.com/design/intelxscale/

[33] RTEMS Home Page, http://www.rtems.com

Chapter 4

A Fast and Accurate Technique for Mapping Parallel Applications on Stream-Oriented MPSoC Platforms with Communication Awareness

4.1 Overview

The problem of allocating and scheduling precedence-constrained tasks on the processors of a distributed real-time system is NP-hard. As such, it has been traditionally tackled by means of heuristics, which provide only approximate or near-optimal solutions. This chapter proposes a complete allocation and scheduling framework, and deploys an MPSoC virtual platform to validate the accuracy of modelling assumptions. The optimizer implements an efficient and exact approach to the mapping problem based on a decomposition strategy. The allocation subproblem is solved through Integer Programming (IP) while the scheduling one through Constraint Programming (CP). The two solvers interact by means of an iterative procedure which has been proven to converge to the optimal solution. Experimental results show significant speedups w.r.t. pure IP and CP exact solution strategies as well as high accuracy with respect to cycle-accurate functional simulation. Two case studies further demonstrate the practical viability of our framework for real-life applications.

4.2 Introduction

Mapping and scheduling problems on multi-processor systems have been traditionally modelled as Integer Linear Programming (IP) problems [2]. In general, even though IP is used as a convenient modelling formalism, there is consensus on the fact that pure IP formulations are suitable only for small problem instances, i.e., applications with a reduced task-level parallelism, because of their high computational cost. For this reason, heuristic approaches are widely used, such as genetic algorithms, simulated annealing and tabu search[17]. However, they do not provide any guarantees on the optimality of the final solution.

On the other hand, complete approaches, which compute the optimal solution at the cost of an increasing computational cost, can be attractive for statically scheduled systems, where the solution is computed once and applied throughout the entire lifetime of the system.

Static allocations and schedules are well suited for applications whose behaviour can be accurately predicted at design time, with minimum run-time fluctuations [38]. This is the case of signal processing applications such as baseband processing, data encryption or video graphics pipelines. Pipelining is a common workload allocation policy to increase throughput of such applications, and this explains why research efforts have been devoted to extending mapping and scheduling techniques to pipelined task graphs[7].

The need to provide efficient solutions to the task-to-architecture mapping problem in reasonable time might lead to symplifying modelling assumptions that can make the problem more tractable. Negligible cache-miss penalties and inter-task communication times, contention-free communication or unbounded on-chip memory resources are examples thereof. Such assumptions however jeopardize the liability of optimizer solutions, and might force the system to work in sub-optimal operating conditions.

In Multi-Processor Systems-on-Chip (MPSoCs) the main source of performance unpredictability stems from the interaction of many concurrent communication flows on the system bus, resulting in unpredictable bus access delays. This also stretches task execution times. Communication architectures should be therefore accurately modelled within task mapping frameworks, so that the correct amount of system-level communication for a given mapping solution can be correctly estimated and compared with the actual bandwidth the bus can deliver. A communication sub-optimal task mapping may lead to reduced throughput or increased latency due to the higher occupancy of system resources. This also has energy implications.

In this chapter we present a novel framework for allocation and scheduling of pipelined task graphs on MPSoCs with communication awareness. We target a general template for distributed memory MPSoC architectures, where each processor has a local memory for fast and energy-efficient access to program data and where messaging support is implemented. A state-of-the-art shared bus is assumed as the system interconnect. Our framework is communication-aware in many senses.

First, we introduce a methodology that determines under which operating conditions system interconnect performance is predictable. In that regime, we derive an accurate high-level model for bus behaviour, which can be used by the optimizer to force a maximum level of bus utilization below which architecture-related uncertainties in system execution are negligible. The limit conditions for predictable bus behaviour are bus protocol-specific, and evolving communication protocols are extending the predictable operating region to higher levels of bus utilization. Our methodology allows system designers to precisely assess when delivered bus bandwidth is lower than the requirements and consequently decide whether to revert to a more advanced system interconnect or to tolerate a comunication-related degradation of system performance.

Second, our mapping strategy discriminates among allocation and scheduling solutions based on the communication cost, while meeting hardware/software constraints (e.g., memory capacity, application real-time requirements).

Our allocation and scheduling framework is based on problem decomposition and combines Artificial Intelligence and Operations Research techniques: the allocation subproblem is solved through Integer Programming (IP), while scheduling through Constraint Programming (CP). However, the two solvers do not operate in isolation, but interact with each other by means of *no-goods* generation, resulting in an iterative procedure which has been proven to converge to the optimal solution. Experimental results show significant speed-ups w.r.t. pure IP and CP exact solution strategies.

Finally, we deploy an MPSoC virtual platform to validate the results of the optimization steps and to more accurately assess constraint satisfaction and objective function optimization. The practical viability of our framework for real-life systems and applications is shown by means of two demonstrators, namely GSM and Multiple-Input-Multiple-Output (MIMO) wireless communication.

The structure of this work is as follows. Section 4.3 illustrates related work. Section 4.4 presents the target architecture while application and system models are reported in Section 4.5. Highlights on Constraint Programming and

Integer Programming are illustrated in Section 5.5. Our combined solver for the mapping problem is described in Section 5.6, its computation efficiency in Section 4.8 and its integration in a software optimization methodology for MP-SoCs in 4.9. Section 4.10 finally shows experimental results.

4.3 Related work

System design methodologies have been investigated for more than a decade, so that now hardware/software codesign has such a rich literature which is impossible to survey exhaustively in one article. This section addresses only the works that are more closely related to the problem and to the class of applications we target. A wider insight on the specific research themes addressed by the HW/SW codesign community over the last decade is reported in [35], while a very comprehensive update on the state of the art in system design can be found in [33].

Mapping and scheduling problems on multi-processor systems have been traditionally modelled as integer linear programming problems, and addressed by means of IP solvers. An early example is represented by the SOS system, which used mixed integer linear programming technique (MILP) [2]. Partitioning with respect to timing constraints has been addressed in [3]. A MILP model that allows to determine a mapping optimizing a trade-off function between execution time, processor and communication cost is reported in [5]. An hardware/software co-synthesis algorithm of distributed real-time systems that optimizes the memory hierarchy (caches) along with the rest of the architecture is reported in [6].

Pipelining is a well known workload allocation policy in the signal processing domain. An overview of algorithms for scheduling pipelined task graphs is presented in [7]. IP formulations as well as heuristic algorithms are traditionally employed. In [9] a retiminig heuristic is used to implement pipelined scheduling, while simulated annealing is used in [10].

Pipelined execution of a set of periodic activities is also addressed in [12], for the case where tasks have deadlines larger than their periods.

The complexity of pure IP formulations for general task graphs has led to the deployment of heuristic approaches (refer to [38] for a comprehensive overview of early results). A comparative study of well-known heuristic search techniques (genetic algorithms, simulated annealing and tabu search) is reported in [17]. Unfortunately, busses are implicit in the architecture. Simulated annealing and tabu search are also compared in [14] for hardware/software partitioning, and minimization of communication cost is adopted as an essential design objective. A scalability analysis of these algorithms for large real-time systems is introduced in [15]. Many heuristic scheduling algorithms are variants and extensions of list scheduling [15]. In general, scheduling tables list all schedules for different condition combinations in the task graph, and are therefore not suitable for control-intensive applications.

The work in [11] is based on Constraint Logic Programming to represent system synthesis problem, and leverages a set of finite domain variables and constraints imposed on these variables. Constraint (Logic) Programming is an alternative approach to Integer Programming for solving combinatorial optimization problems [17]. Both techniques can claim individual successes but practical experience indicates that neither approach dominates the other in terms of computational performance on problems similar to the one faced in this chapter. The development of a hybrid CP-IP solver that captures the best features of both would appear to offer scope for improved overall performance. However, the issue of communication between different modelling paradigms arises. One method is inherited from the Operations Research and is known as Benders Decomposition [24]: it is an iterative solving strategy that has been proven to converge producing the optimal solution. Benders Decomposition has been extended, and called Logic-Based Benders Decomposition in [7], for dealing with any kind of solver, like a CP solver. There are a number of papers using Benders Decomposition in a CP setting[18][19][6][8].

In this work, we take the Logic-Based Benders Decomposition approach, and come up with original design choices to effectively apply it to the context of MPSoCs. We opt for decomposing the mapping problem in two sub-problems: (i) mapping of tasks to processors and of data to memories and (ii) scheduling of tasks in time on their execution units. We tackle the mapping sub-problem with IP and the scheduling one with CP, and combine the two solvers in an iterative strategy which converges to the optimal solution [7]. Our problem formulation will be compared with the most widely used traditional approaches, namely CP and IP modelling of the entire mapping and scheduling problem as a whole, and the significant cut down on search time that we can achieve is proved. Moreover, in contrast to most previous work, the results of the optimization framework and its modelling assumptions are validated by means of cycle-accurate functional simulation on a virtual platform.

4.4 Target Architecture

Our mapping strategy targets a general architectural template for a message-oriented distributed memory MPSoC. The distinctive features of this template include: (i) support for message exchange between parallel computation subsystems, (ii) availability of local memory devices at each computation sub-

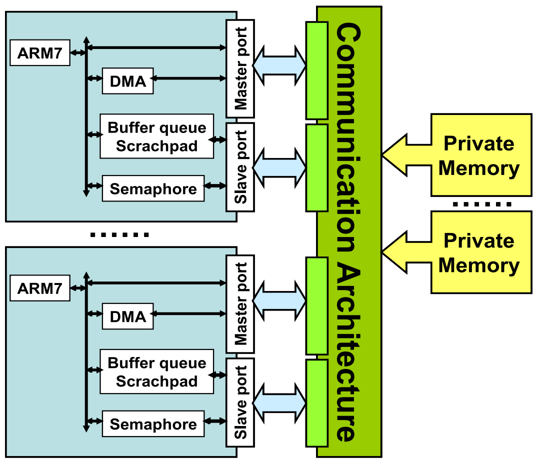


Figure 4.1: Message-oriented distributed memory architecture.

system and of non-local (i.e., accessible through the system bus) memories to store program data exceeding local memory size. The remote storage can be provided by a unified memory with partitions associated with each processor or by a separate private memory for each processor core connected to the system bus. This assumption concerning the memory hierarchy reflects the typical trade-off between low access cost, low capacity local memory devices and high cost, high capacity memory devices at a higher level of the hierarchy. Several MPSoC platforms available on the market match our template, such as the Cell Processor[8], the Silicon Hive Avispa-CH1 processor[31], the Cradle CT3600 family of multiprocessor DSPs[30] or the ARM11 MPCore platform[29].

The only restriction that we pose in the template concerns the communication queues, which are assumed to be single-token. Therefore, in a producer-consumer pair, each time a data unit is output by the producer, the consumer has to read it before the producer can run again, since it has its single-entry output queue occupied. The extension of our framework to multi-token queues is left for future work and can be seen as an incremental improvement of the optimization framework.

We modelled one instance of this architectural template in order to test our optimization framework (see Fig. 5.1). The computation sub-systems are supposed to be homogeneous and consist of ARM7 cores (including instruction and data caches) and of tightly coupled software-controlled scratchpad memories for fast access to program operands and for storing input data. We used an AMBA AHB[4] bus as shared system interconnect.

In our implementation, hardware and software support for efficient messaging is provided. Messages can be directly moved between scratchpad memories. In order to send a message, a producer core writes in the message queue stored in its local scratchpad memory, without generating any traffic on the interconnect. After the message is ready, the consumer can transfer it to its own scratchpad or to a private memory space. Data can be transferred either by the processor itself or by a direct memory access controller, when available. In order to allow the consumer to read from the scratchpad memory of another processor, the scratchpad memories should be connected to the communication architecture also by means of slave ports, and their address space should be visible to the other processors.

As far as synchronization is concerned, when a producer intends to generate a message, it checks a local semaphore which indicates whether the queue is empty or not. When a message can be stored, its availability is signaled to the consumer by releasing its local semaphore through a single write operation that goes through the bus. Semaphores are therefore distributed among the processing tiles, resulting in two advantages: the read/write traffic to the semaphores is distributed and the producer (consumer) can locally poll whether space (a message) is available, thereby reducing bus traffic.

Furthermore, our semaphores may interrupt the local processor when released, providing an alternative mechanism to polling. In fact, if the semaphore is not available, the polling task registers itself on a list of tasks waiting for that semaphore and suspends itself. Other tasks on the processor can then execute. As soon as the semaphore is released, it generates an interrupt and the corresponding service routine reactivates all tasks on the waiting list.

A DMA engine is attached to each core, as presented in [25], allowing efficient data transfers between the local scratchpad and non-local memories reachable through the bus. The DMA control logic supports multichannel programming, while the DMA transfer engine has a dedicated connection to the scratchpad memory allowing fast data transfers from or to it.

Finally, each processor core has a private memory, which can be accessed only by gaining bus ownership. This memory could be on-chip or off-chip depending on the specific platform instantiation. It has a higher access cost and can be used to store program operands that do not fit in scratchpad memory. Optimal memory allocation of task program data to the scratchpad versus the private memory is a specific goal of our optimization framework, dealing with the constraint of limited size of local memories in on-chip multi-processors.

The software support is provided by a real-time multi-processor operating system called RTEMS [11] and by a set of high-level APIs to support message passing on the considered distributed memory architecture. The communication and synchronization library abstracts low level architectural details to the programmer, such as memory maps or explicit management of hardware semaphores[32].

Our implementation thus supports: (i) processor or DMA-initiated memory-to-memory transfers, (ii) polling-based or interrupt-based synchronization, and (iii) flexible allocation of the consumer's message buffer to the local scratchpad or the non-local private memory.

4.5 High-level application and system models

4.5.1 Task model

Our mapping methodology requires to model the multi-task application to be mapped and executed on top of the target hardware platform as a Directed Acyclic Task Graph with precedence constraints. In particular, we focus on pipelined task graphs, representative of signal processing workloads. A real-time requirement is typically specified for this kind of applications, consisting for instance of a minimum required throughput for the pipeline of tasks. Tasks are the nodes of the graph and edges connecting any two node indicate task dependencies. Computation, storage and communication requirements should be annotated onto the graph as follows.

The task execution time is given in two cases: program data is stored entirely in scratchpad memory and local data is stored in remote private memory only. In this latter case, the impact of cache misses on execution time is taken into account.

Our application model associates three kinds of memory requirements to each task:

- Program Data: storage space is required for computation data and for processor instructions. They can be allocated by the optimizer either on the local scratchpad memory or on the remote private memory.

- Internal State: when needed, an internal state of the task can be stored either locally or remotely.

- Communication queues: the task needs communication queues to store outgoing as well as incoming messages to/from other tasks. For the sake of efficient messaging, we pose the constraint that such communication queues should be stored in local scratchpad memory only. So, allocation of these queues is not a degree of freedom for the optimizer.

We assume that application tasks initially check availability of input data and of space for writing computation results (i.e., the output queue must have been freed by the downstream task), in an SDF-like (synchronous dataflow) semantics. Actual input data transfer and task execution occur only when both conditions are met. These assumptions simply result in an atomic execution of the communication and computation phases of each task, thus avoiding the

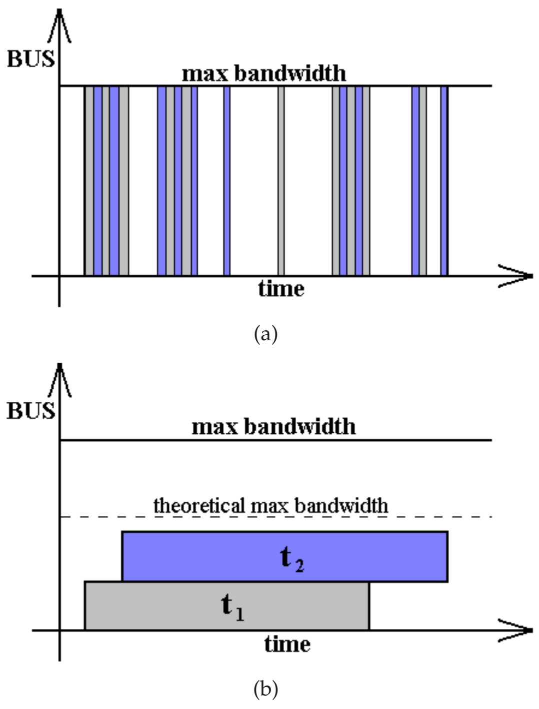


Figure 4.2: (a) Bus allocation in a unary model; (b) Bus allocation in a coarse-grain additive model

need to schedule communication as a separate task.

4.5.2 Bus model

Whenever predictable performance is needed for time-critical applications, it is important to avoid high levels of congestion on the bus, since this makes completion time of bus transactions (and hence of task execution) much less predictable. Average or peak bus bandwidth utilization can be modulated by means of a proper communication-aware task mapping strategy.

When the bus is required to provide a cumulative bandwidth from concurrently executing tasks that does not exceed a certain threshold, its behaviour can be accurately abstracted by means of a very simple *additive model*. In other words, the bus delivers an overall bandwidth which is approximatively equal to the sum of the bandwidth requirements of the tasks that are concurrently making use of it.

This model, provided the working conditions under which it holds are carefully delimited, has some relevant advantages with respect to the scheduling problem model. First, it allows to model time at a coarse granularity. In fact, busses rely on the serialization of bus access requests by re-arbitrating on a transaction basis. Modelling bus allocation at such a fine granularity would make the scheduling problem overly complex since it should be modelled as a unary resource (i.e., a resource with capacity one). In this case, task execution should be modelled using the clock cycle as the unit of time and the resulting scheduling model would contain a huge number of variables. The additive model instead considers the bus as an additive resource, in the sense that more activities can share bus utilization using a different fraction of the total bus bandwidth. Fig. 4.2(a) illustrates this assumption. The figure represents the bus allocation and scheduling in a real processor, where the bus is assigned to different tasks at different times on a transaction-per-transaction basis. Each task, when owning the bus, uses its entire bandwidth.

Fig. 4.2(b), instead, represents how we model the bus, abstracting away the transaction-based allocation details. We assume that each task consumes a fraction of the bus bandwidth during its execution time. Note that we have two thresholds: the maximum bandwidth that the bus is physically able to deliver, and the theoretical one beyond which the additive model fails to predict the interconnect behaviour because of the impact of contention. We will derive this latter threshold in the experimental section by means of extensive simulation runs.

In order to define the fraction of the bus bandwidth absorbed by each task, we consider the amount of data they have to access from their private memories and we spread it over its execution time. In this way we assume that the task is uniformly consuming a fraction of the bus bandwidth throughout its execution time. This assumption will be validated in presence of different traffic patterns in the experimental section.

Another important effect of the bus additive model is that task execution times will not be stretched as an effect of busy waiting on bus transaction completion. Once the execution time of a task is characterized in a congestion free regime, it will be only marginally affected by the presence of competing bus access patterns, in the domain where the additive model holds.

Mapping tasks in such a way that the bus utilization lies below the additive threshold forces the system to make efficient use of available bandwidth. However, our methodology can map tasks to the system while meeting any requirement on bus utilization. Therefore, if a given application cannot be mapped with the bus working in the additive regime, it is on burden of the designer to choose whether to increase maximum allowable peak bus utilization (at the cost of a lower degree of confidence in optimizer performance predictions) or to revert to a more advanced system interconnect. Even in the first case, our methodology helps designers to map their applications with minimum additive threshold crossing.

4.6 Background on optimization techniques

In this section, we recall the basic concepts behind the method we use in this chapter, namely the Logic Based Benders Decomposition, and the two optimization techniques we use for solving each subproblem resulting from the decomposition, namely Constraint Programming and Integer Programming.

4.6.1 Logic Based Benders Decomposition

The technique we use in this chapter is derived from a method, known in Operations Research as Benders Decomposition [24], and refined by [7] with the name of Logic-based Benders Decomposition. The classical Benders Decomposition method decomposes a problem into two loosely connected subproblems. It enumerates values for the connecting variables. For each set of enumerated values, it solves the subproblem that results from fixing the connecting variables to these values. The solution of the subproblem generates a Benders cut that the connecting variables must satisfy in all subsequent solutions enumerated. The process continues until the master problem and subproblem converge providing the same value. The classical Benders approach, however, requires that the subproblem be a continuous linear or nonlinear programming problem. Scheduling is a combinatorial problem that has no practical linear or nonlinear programming model. Therefore, the Benders decomposition idea can be extended to a logic-based form (Logic Based Benders Decomposition - LBBD) that accommodates an arbitrary subproblem, such as a discrete scheduling problem. More formally, as introduced in [7], a problem can be written as

$$\min f(y) \tag{4.1}$$

$$s.t. \quad p_i(y)\ i \in I_1 \ \text{Master Problem Constraints} \tag{4.2}$$

$$g_i(x)\ i \in I_2 \ \text{Subproblem Constraints} \tag{4.3}$$

$$q_i(y) \rightarrow h_i(x)\ i \in I_3 \ \text{Conditional Constraints} \tag{4.4}$$

$$y \in Y \ \text{Master Problem Variables} \tag{4.5}$$

$$x_j \in D_i \ \text{Subproblem Variables} \tag{4.6}$$

We have master problem constraints, subproblem constraints and conditional constraints linking the two models. If we solve the master problem to optimality, we obtain values for variables y in I_1, namely $\bar{y}$ and the remaining problem is a feasibility problem:

$$g_i(x)\ i \in I_2 \ \text{ Subproblem Constraints} \tag{4.7}$$

$$q_i(\bar{y}) \rightarrow h_i(x)\ i \in I_3 \ \text{ Conditional Constraints} \tag{4.8}$$

$$x_j \in D_i \ \text{ Subproblem Variables} \tag{4.9}$$

We can add to this problem a secondary objective function, say $f_1(x)$ just to discriminate among feasible solutions. If the problem is infeasible, a Benders cut $B_y(y)$ is created constraining variables y. The master problem thus becomes

$$\min f(y) \tag{4.10}$$

$$s.t. \quad p_i(y)\ i \in I_1 \ \text{ Master Problem Constraints} \tag{4.11}$$

$$B_{y_i}(y)\ i \in 1..h \ \text{ Benders cuts} \tag{4.12}$$

$$y \in Y \ \text{ Master Problem Variables} \tag{4.13}$$

y_i is the solution found at iteration i of the master problem.

In practice, to avoid the generation of master problem solutions that are trivially infeasible for the subproblem, it is worth adding a relaxation of the subproblem to the master problem.

Deciding to use the LBBD to solve a combinatorial optimization problem implies a number of design choices that strongly affect the overall performance of the algorithm. Design choices are:

- how to decompose the problem, i.e., which constraints are part of the master problem and which instead are part of the subproblem. This influences the objective function and its dependency on master and subproblem variables;

- which solver to choose for each decomposition: not all problems are solved effectively by the same solver. We consider in this chapter Constraint and Integer Linear programming that cover a variety of optimization problems effectively;

- which model to use for feeding each solver: given the problem and the solver we still need to design the problem model, i.e., variables, constraints and objective function. In combinatorial optimization, a wrong model results always in poor solver performances;

- which Benders cuts to use, establishing the interaction between the master and the subproblem;
- which relaxation to use so as to avoid the generation of trivially infeasible solutions in the master problem.

In the following we provide preliminaries on Constraint Programming and Integer Programming, while in section 5.6 we detail the design choices performed for the mapping and scheduling problem at hand.

4.6.2 Constraint Programming

Constraint Programming (CP) has been recognized as a suitable modelling and solving tool to face combinatorial (optimization) problems. The CP modeling and solving activity is highly influenced by the Artificial Intelligence area on Constraint Satisfaction Problems, CSPs (see, e.g., the book by [37]). A CSP is a triple $\langle V, D, C\rangle$ where V is a set of variables $X_1, \ldots, X_n$, D is a set of finite domains $D_1, \ldots, D_n$ representing the possible values that variables can assume, and C is a set of constraints $C_1, \ldots, C_k$. Each constraint involves a set of variables $V' \subseteq V$ and defines a subset of the cartesian product of the corresponding domains containing feasible tuples of values. Therefore, constraints limit the values that variables can simultaneously assume. A solution of a CSP is an assignment of values to variables which is consistent with constraints.

Constraints can be either mathematical or symbolic. Mathematical constraints have the form: $t_1\ R\ t_2$ where t_1 and t_2 are finite terms, i.e., variables, finite domain objects and usual expressions, and R is one of the constraints defined on the domain of discourse (e.g., for integers we have the usual relations: $>, \geq, <, \leq, =, \neq$). For example, if two activities i and j characterized by starting times $Start_i$ and $Start_j$ and durations d_i and d_j are linked by a precedence constraint stating that activity i should be executed before activity j, the following mathematical constraint can be imposed, $Start_i + d_i \leq Start_j$. Symbolic constraints, called also global constraints, are predicates involving finite domain variables. They are expressive and powerful constraints (which can also be defined by the user) embedding constraint-dependent filtering algorithms. A typical global constraint is the

$$\textit{alldifferent}([X_1, \ldots, X_n])$$

available in most CP solvers. Declaratively, the constraint *alldifferent*$([X_1, \ldots, X_n])$ holds iff all variables are assigned to a different value. Thus, it is declaratively equivalent to a set of $n * (n-1)/2$ binary inequality constraints. However, its compact representation allows more concise models and embeds a specialized efficient graph-based filtering algorithm [36]. Many constraints have been devised for scheduling, which is the most successful application of Constraint Programming. In particular, many kinds of resource and temporal constraints have been devised so as to solve large problem instances, see [16]. As an example, let us consider the cumulative constraint used for modelling limited resource availability in scheduling problems. Its parameters are: a list of variables $[S_1, \ldots, S_n]$ representing the starting time of all activities sharing the resource, their duration $[D_1, \ldots, D_n]$, the resource consumption for each activity $[R_1, \ldots, R_n]$ and the available resource capacity C. Clearly this constraint holds if in any time step where at least one activity is running the sum of the required resource is less than or equal to the available capacity. The constraint $cumulative([S_1, \ldots, S_n], [D_1, \ldots, D_n], [R_1, \ldots, R_n], C)$ holds iff

$$\forall j \sum_{S_j \le i < S_j + D_j} R_i \le C$$

4.6.3 Integer Programming

Another solution technique, which is well known and widely used in the system design community is Integer Programming (IP). Integer programming is an older method, with roots that date back to the late 1950s. Integer Programming can be thought of as a restriction of Constraint Programming. In fact, Integer Programming has only two types of variables: integer variables whose domain contain non-negative integers and continuous variables whose domain contain non-negative real values. In addition, IP allows only one type of constraint: linear inequalities. Finally, the objective function must be linear in the variables. It seems that these restrictions make integer programming much narrower than constraint programming. However, many problems can still be modeled effectively, and algorithms for integer programs can find optimal solutions quickly for many applications. The solving principle of IP is based on the solution of the *linear relaxation*, allowing arbitrary sets of linear constraints to be treated as a global constraint, providing a global view of the problem. The relaxation provides a bound enabling efficient pruning of the search tree and directing search toward promising regions.

The standard form of an IP is the following: let x be the vector of variables, $x = [x_1, x_2, \ldots, x_n]$. A set of these variables I are required to take on integer values, while the remaining variables can take on any real value. Each variable can have a range, represented by vectors l and u such that $l_i \le x_i \le u_i$. A *linear constraint* on the variables is a vector of coefficients $a = [a_1, \ldots, a_n]$ and a scalar right-hand-side b. The constraint is then the requirement that

$$\sum_j a_j x_j = b$$

The "=" in the constraint can also be $\leq$ or $\geq$ (but not $<$ or $>$). The objective function is formed by a vector of coefficients $c = [c_1, c_2, \ldots, c_n]$, with the objective of minimizing (or maximizing) cx. An integer program consists of a single linear objective and a set of constraints. If we create a matrix $A = [a_{ij}]$, where a_{ij} is the coefficient for variable j in the ith constraint, then an integer program can be written:

$$\min cx \tag{4.14}$$

$$s.t. \quad Ax = b \tag{4.15}$$

$$l \leq x \leq u \tag{4.16}$$

$$x_j \text{ integer for all } j \in I \tag{4.17}$$

For many applications, it is worth working within the limits of integer programming to achieve high performance.

4.7 Model definition

The two main approaches followed by the system design community when facing software mapping problems in MPSoCs are: (1) either modelling and solving the problem to optimality as an Integer Program whatever the problem structure is or (2) using a special purpose heuristic algorithm requiring sophisticated debugging and tuning and achieving sub-optimal solutions. In this chapter, we claim that:

- Whenever allocation and scheduling can be performed off-line due to the intrinsic features of the application (predictable workload), the correct approach is to solve these problems to optimality, since their solution is computed once for all at design time and applied during the entire lifetime of the system. Optimal solutions enable to achieve significant performance speed-ups.
- Analyzing and exploiting the problem structure helps in choosing the best solving technique. Integer Programming is an effective solving framework but it is not always the best technique one can use. Constraint Programming effectively deals with fine time granularities, temporal con-

straints, resource constraints, and different kind of activities. In general, the best solution strategy can be applied to each subproblem structure.

We have first tried to solve the overall problem (mapping and scheduling) to optimality using a single approach. We have tested both Constraint Programming alone and Integer Programming alone on the problem without success. Therefore, we have switched to Logic Based Benders Decomposition. As shown in section 5.5.3, a number of design choices should be addressed.

- **How to decompose the problem.** We split the overall mapping problem into two sub-problems: (1) the allocation of tasks to processors and memory requirements to storage devices, trying to minimize the communication cost, and (2) the scheduling sub-problem, where the minimization of execution time (or makespan) can be chosen as secondary design objective.

  Given the critical role played by on-chip communication in determining performance predictability of highly integrated MPSoCs, we select communication cost minimization as the objective function of the overall problem. This function involves only variables of the first problem. In particular, we have a communication cost each time two communicating tasks are allocated on different processors, and each time a memory slot is allocated on a remote memory device. Once we have optimally allocated tasks to resources, we can minimize the global schedule makespan.

  Note that our decomposition choice is, to our knowledge, original. Other approaches to allocation and scheduling [6] [8] cope with scheduling problems where tasks assigned to different machines are not linked by any constraint. Therefore, the subproblem is composed by a set of independent single machine scheduling problems.

  Different objective functions can be easily supported by our technique. Clearly, one should change the relaxation of the subproblem and the no-goods. The aim of this work is not to prove the effectiveness of Logic-Based Benders Decomposition in general, but specifically for the problem at hand.

- **Which solver to choose for each decomposition.** There are no general guidelines for choosing the best solver for the problem at hand. Indeed, it is not always possible to choose the best solver for a given problem instance. For some problems, it is widely recognized that either Integer Programming or Constraint Programming are the techniques of choice. Integer Programming is effective for coping with optimization problems, it has a global problem view due to the use of linear relaxations, but sometimes its models are too large and somewhat unnatural. On the other hand, Constraint Programming has an effective way to cope with the so called *feasibility reasoning*, encapsulating efficient and incremental filtering algorithms into global constraints. However, CP has a naive way to cope with optimization problems by successively solving a set of constraint satisfaction problems with tighter bounds on the objective function.

For the problem at hand, the allocation problem has been solved via Integer Programming. It better copes with objective functions based on the sum of assignment costs. For the scheduling problem, the solver is instead based on Constraint Programming since it better copes with temporal resource constraints and finer time granularity.

- **Which model to use for feeding each solver.** This part will be described in detail in the next sections. In particular, the allocation problem model is described in section 4.7.1 while the scheduling problem model is described in section 4.7.2.
- **Which Benders cuts to use.** This aspect is essential for the interaction between the two solvers. We solve the allocation problem first (called master problem), and the scheduling problem (called subproblem) later. The master is solved to optimality and its solution passed to the subproblem solver. If the solution is feasible, then the overall problem is solved to optimality, since the main objective function depends only on master problem variables. If, instead, the master solution cannot be completed by the subproblem solver, a no-good is generated and added to the model of the master problem, roughly stating that the solution passed should not be recomputed again (it becomes infeasible), and a new optimal solution is found for the master problem respecting the (set of) no-good(s) generated so far. Being the allocation problem solver an IP solver, the no-good has the form of a linear constraint.
- **Which relaxation to use.** Now let us note the following: the assignment problem allocates tasks to processors, and memory requirements to storage devices minimizing communication costs. However, since real-time constraints are not taken into account by the allocation module, the solution obtained tends to pack all tasks in the minimal number of processors. In other words, the only constraint that prevents to allocate all tasks to a single processors is the limited capacity of the tightly coupled memory devices. However, these trivial allocations do not consider throughput constraints which make them most probably infeasible for the overall problem. To avoid the generation of these (trivial) assignments, we

should add to the master problem model a relaxation of the subproblem. In particular, we should state in the master problem that the sum of the durations of tasks allocated to a single processor does not exceed the real time requirement. In this case, the allocation is far more similar to the optimal one for the problem at hand. The use of a relaxation in the master problem is widely used in practice and helps in producing better solutions.

4.7.1 Allocation problem model

The allocation problem is the problem of allocating n tasks to m processors and memory requirements to storage devices. The objective function is the minimization of the amount of data transferred on the bus. We solve the allocation problem using an IP model. We consider four decision variables: T_{ij}, taking value 1 iff task i executes on processor j; Y_{ij}, taking value 1 iff task i allocates the program data on the scratchpad memory of processor j; Z_{ij}, taking value 1 iff task i allocates the internal state on the scratchpad memory of processor j; X_{ij}, taking value 1 iff tasks i and $i+1$ execute on different processors, one of them being processor j, therefore the 2 tasks communicate using the bus. Variables X_{ij} have only two indexes since we are considering a pipeline, where a task i communicates only with the task $i+1$. When modelling a general task graph these variables must have the form X_{ikj}, taking value 1 iff two communicating tasks i and k execute on different processors, one of them being processor j. The linear constraints introduced in the model are:

$$\sum_{j=1}^{m} T_{ij} = 1, \forall i \in 1 \dots n \tag{4.18}$$

$$T_{ij} + T_{i+1j} + X_{ij} - 2K_{ij} = 0 \ , \forall i \in 1 \dots n \ , \forall j \tag{4.19}$$

Constraints (4.18) state that each process can execute only on a processor, while constraints (4.19) state that X_{ij} can be equal to 1 iff $T_{ij} \neq T_{i+1j}$, that is, iff task i and task $i+1$ execute on different processors. K_{ij} are integer binary variables forcing the sum $T_{ij}+T_{i+1j}+X_{ij}$ to be either 0 or 2 (in fact, X_{ij} is the `exor` of T_{ij} and T_{i+1j}). We also add to the model the constraints stating that if a task i does not execute on a processor j, it cannot allocate its program data or its internal state in the local scratchpad of processor j, i.e. $T_{ij} = 0 \Rightarrow Y_{ij} = 0, Z_{ij} = 0$. For each group of consecutive tasks whose execution times sum exceeds the RT requirement, we introduce in the model a constraint preventing the solver to allocate all the tasks in the group to the same processor. To generate these constraints, we find out all groups of consecutive tasks whose execution times sum exceeds RT. Constraints are the following:

$$\sum_{i \in S} Dur_i > RT \Rightarrow \sum_{i \in S} T_{ij} \leq |S| - 1 \; \forall j \tag{4.20}$$

This is a relaxation of the scheduling problem, added to the master problem to prevent the generation of trivially infeasible solutions. The objective function is the minimization of the communication cost, i.e., the total amount of data transferred on the bus for each pipeline iteration. Contributions to the communication cost arise when a task allocates its program data and/or internal state to the remote memory, and when two consecutive tasks execute on different processors, and their communication messages must be transferred through the bus from the communication queue of one processor to that of the other one. Using the decision variables described above, we have a contribution respectively when: $T_{ij} = 1, Y_{ij} = 0$, or $T_{ij} = 1, Z_{ij} = 0$, or $X_{ij} = 1$. Therefore, the objective function is to minimize:

$$\sum_{j=1}^{m} \sum_{i=1}^{n} \Big(Mem_i (T_{ij} - Y_{ij}) + 2 \times State_i (T_{ij} - Z_{ij}) +$$
$$+ (Data_i X_{ij})/2) \Big) \tag{4.21}$$

where Mem_i, $State_i$ and $Data_i$ are the amount of data used by task i to store respectively the program data, the internal state and the communication queue.

4.7.2 Scheduling problem model

Once tasks have been allocated to the processors, we need to schedule process execution. Since we are considering a pipeline of tasks, we need to analyze the system behavior at working rate, that is when all processes are running or ready to run. To do that, we consider several instantiations of the same process; to achieve a working rate configuration, the number of repetitions of each task must be at least equal to the number of tasks n; in fact, after n iterations, the pipeline is at working rate. So, to solve the scheduling problem, we must consider at least n^2 tasks (n iterations for each process), see Fig. 4.3.

In the scheduling problem model, for each task $Task_{ij}$ (the j-th iteration of the i-th process) we introduce a variable A_{ij}, representing the computation activity of the task. Once the allocation problem is solved, we statically know if a task needs to use the bus to communicate with another task, or to read/write computation data and internal state from the remote memory. In particular, each activity A_{ij} must read the communication queue from the activity A_{i-1j}, or from the pipeline input if $i = 0$. For this purpose, we introduce in the model

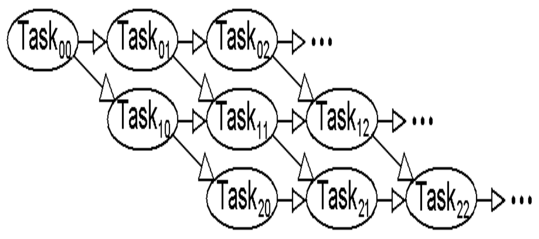


Figure 4.3: Precedence constraints among the activities

the activities In_{ij}. If a process requires an internal state, the state must be read before the execution and written after the execution: we therefore introduce in the model the activities RS_{ij} and WS_{ij} for each process i requiring an internal state. The durations of all these activities depend on whether data are stored in the local or the remote memory but, after the allocation, these times can be statically estimated. Fig. 4.3 depicts the precedence constraints among tasks. The horizontal arcs (between $Task_{ij}$ and $Task_{i,j+1}$) represent just precedence constraints, while the diagonal arcs (between $Task_{ij}$ and $Task_{i+1,j}$) represent precedences due to communication and are labelled with the amount of data to communicate. Each task $Task_{ij}$ is composed by activity A_{ij} possibly preceded by the internal state reading activity RS_{ij}, and input data reading activity In_{ij}, and possibly followed by the internal state writing activity WS_{ij}. The precedence constraints among the activities are:

$$A_{i,j-1} \prec In_{ij} \ , \ \forall\, i,j \tag{4.22}$$

$$In_{ij} \prec A_{ij} \ , \ \forall\, i,j \tag{4.23}$$

$$A_{i-1,j} \prec In_{ij} \ , \ \forall\, i,j \tag{4.24}$$

$$RS_{ij} \preceq A_{ij} \ , \ \forall\, i,j \tag{4.25}$$

$$A_{ij} \preceq WS_{ij} \ , \ \forall\, i,j \tag{4.26}$$

$$In_{i+1,j-1} \prec A_{ij} \ , \ \forall\, i,j \tag{4.27}$$

$$A_{i,j-1} \prec A_{ij} \ , \ \forall\, i,j \tag{4.28}$$

where the symbol $\prec$ means that the activity on the right should follow the activity on the left, and the symbol $\preceq$ means that the activity on the right must start as soon as the execution of the activity on the left completes: i.e., $A \prec B$ means $Start_A + Dur_A \leq Start_B$, and $A \preceq B$ means $Start_A + Dur_A = Start_B$. Constraints (4.22) state that each process iteration can start reading the communication queue only after the end of its previous iteration: a task needs to access the data stored in the communication queue during its whole execution, so the memory storing these data can only be freed when the computation activity A_{ij} ends. Constraints (4.23) state that each task can start computing only when it has read the input data, while constraints (4.24) state that each task can read the input data only when the previous task has generated them. Constraints (4.25) and (4.26) state that each task must read the internal state just before the execution and write it just afterwards. Constraints (4.27) state that each task can execute only if the previous iteration of the following task has read the input data; in other words, it can start only when the data stored in its communication queue has been read by the target process. Constraints (4.28) state that the iterations of each task must execute in order. We also introduced the real-time requirement constraints $Start(A_{ij}) - Start(A_{i,j-1}) \leq RT \ , \ \forall\, i, j$, whose relaxation is used in the allocation problem model. The time elapsing between two consecutive executions of the same task can be at most RT. Processors are modelled as unary resources, stating that only one activity at a time can execute on each processor, while the bus is modelled as a shared resource (see subsection 4.5.2): several activities can share the bus, each of them consuming a fraction of the total bandwidth; a $cumulative$ constraint is introduced ensuring that the total bus bandwidth consumption (or a lower threshold) is never exceeded.

4.8 Computational efficiency

To test the computational efficiency of our approach, we now compare the results obtained using this model (**Hybrid** in the following) with results obtained using only a CP or IP model to solve the overall problem to optimality. Actually, since the first experiments showed that both CP and IP approaches are not able to find even the first solution, except for the easiest instances, within 15 minutes, we simplified these models removing some variables and constraints. In CP, we fixed the activities execution time not considering the execution time variability due to remote memory accesses, therefore we do not consider the In_{ij}, RS_{ij} and WS_{ij} activities, including them statically in the activities A_{ij}. In IP, we do not consider all the variables and constraints involving the bus: we do not model the bus resource and we therefore suppose that each activity can access data whenever it is necessary.

We generated a large variety of problems, varying both the number of tasks and processors. All the results presented are the mean over a set of 10 instances for each task or processor number. All problems considered have a solution. Experiments were performed on a 2GHz Pentium 4 with 512 Mb RAM and leveraged state-of-the-art professional solving tools, namely ILOG CPLEX 8.1, ILOG Solver 5.3 and ILOG Scheduler 5.3.

In Fig. 4.4 we compare the algorithms search time for problems with a different number of tasks and processors respectively. Times are expressed in seconds and the y-axis has a logarithmic scale.

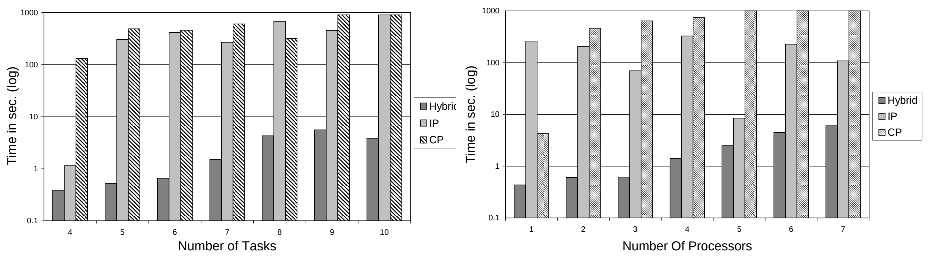


Figure 4.4: Comparison between algorithms search times for different task number (left) and for different processor number (right)

Although CP and IP deal with a simpler problem model, we can see that these algorithms are not comparable with Hybrid, except when the number of tasks and processors is low and the problem instance is very easy to be solved, and Hybrid incurs the communication overhead between two models. As soon as the number of tasks and/or processors grows, IP and CP performance worsen and their search times become orders of magnitude higher w.r.t. Hybrid. Furthermore, we considered in the figures only instances where the algorithms are able to find the optimal solution within 15 minutes, and, for problems with 6 tasks or 3 processors and more, IP and CP can find the solution only in the 50% or less of the cases, while Hybrid can solve 100% of the instances. We can see in addition, that Hybrid search time scales up linearly in the logarithmic scale.

We also measured the number of times the CP and IP solvers iterate. We found that, due to the limited size of the scratchpad and to the relaxation of the sub-problem added to the master, the solver iterates always 1 or 2 times. Removing the relaxation, it iterates up to 15 times. This result gives evidence that, in a Benders decomposition based approach, it is very important to introduce a relaxation of the sub-problem in the master, and that the relaxation we use is very effective although very simple.

4.9 Validation methodology

In this section we explain how to deploy our optimization framework in the context of a real system-level design flow. Our approach consists of using a virtual platform to pre-characterize the input task set, to simulate the allocation and scheduling solutions provided by the optimizer and to detect deviations of measured performance metrics with respect to predicted ones.

For each task in the input graph we need to provide the following information: bus bandwidth requirement for reading input data in case the producer runs on a different processor, time for reading input data if the producer runs on the same processor, task execution time with program data in scratchpad memory, task execution overhead due to cache misses when program data resides in remote private memory. For each pipelined task graph, this information can be collected with $2 + N$ simulation runs on the MPARM simulator[26], where N is the number of tasks. Recall that this is done once for all. We model task communication and computation separately to better account for their requirement on bus utilization, although from a practical viewpoint they are part of the same atomic task. The initial communication phase consumes a bus bandwidth which is determined by the hardware support for data transfer (DMA engines or not) and by the bus protocol efficiency (latency for a read transaction). The computation part of the task instead consumes an average bandwidth defined by the ratio of program data size (in case of remote mapping) and execution time. A less accurate characterization framework can be used to model the task set, though potentially incurring more uncertainty with respect to optimizer's solutions. We use the virtual platform also to calibrate the bus additive model, specifying the range where this model holds. For an AMBA AHB bus, we found that tasks should not concurrently ask for more than 50% of the theoretical bandwidth the bus can provide (400 MByte/sec with 1 wait state memories), otherwise congestion causes a bandwidth delivery which does not keep up with the requirements.

The input task parameters are then fed to the optimization framework, which provides optimal allocation of tasks and memory locations to processor and storage devices respectively, and a feasible schedule for the tasks meeting the real-time requirements of the application. Two options are feasible at this point. First, the optimizer uses the conservative maximum bus bandwidth indicated by the virtual platform, and the derived solutions are guaranteed to be accurate (see section 4.10). Second, the optimizer uses a higher bandwidth than specified, in order to improve bus utilization, and the virtual platform must then be used to assess the accuracy of the optimization step (e.g., constraint satisfaction, validation of execution and data transfer times). If the accuracy is not satisfactory, a new iteration of the procedure will allow to progressively decrease the maximum bandwidth until the desired level of accuracy is reached with the simulator.

Note that the scheduler of the RTEMS operating system allows to implement all the scheduling solutions provided by the optimizer. For the case we are considering (stream-oriented processing with single token communication among the pipeline stages) it can be proven that all schedules are periodic. The interested reader can read the proof in Appendix I. Our framework assumes that no preemption nor time-slicing is implemented by the OS. Most schedules generated by the optimizer can be implemented by means of priority-based scheduling, but not all of them. For those remaining cases, RTEMS provides scheduling APIs with which one task can decide which task to activate next. In this way, all possible schedules can be implemented.

4.10 Experimental Results

We have performed three kinds of experiments, namely (i) validation and calibration of the bus additive model, (ii) measurement of deviations of simulated throughput from the one computed by the optimizer on a large number of problem instances, (iii) experiments devoted to show the viability of the proposed approach by means of two demonstrators.

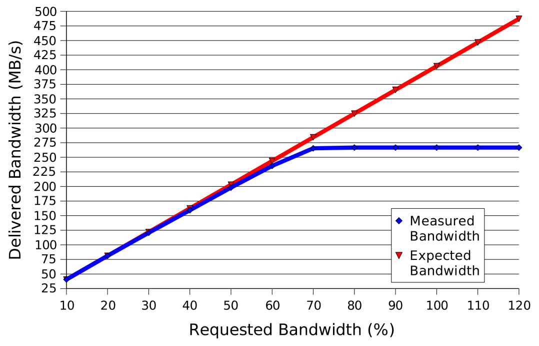


Figure 4.5: Implications of the bus additive model

4.10.1 Validation of the bus additive model

The behaviour of the bus additive model is illustrated by the experiment of Fig.4.5. An increasing number of AMBA-compliant uniform traffic generators, consuming each 10% of the maximum theoretical bandwidth (400 MByte/sec), have been connected to the bus, and the resulting real bandwidth provided by the bus measured in the virtual platform. It can be clearly observed that the delivered bandwidth keeps up with the requested one until the sum of the requirements amounts to 60% of the maximum theoretical bandwidth. This defines the actual maximum bandwidth, notified to the optimizer, under which the bus works in a predictable way. If the communication requirements exceed the threshold, as a side effect we observe an increase of the execution times of running tasks with respect to those measured without bus contention, as depicted in Fig.4.6. For this experiment, synthetic tasks running on each processor have been employed. The 60% bandwidth threshold value corresponds to an execution time variation of about 2% due to longer bus transactions.

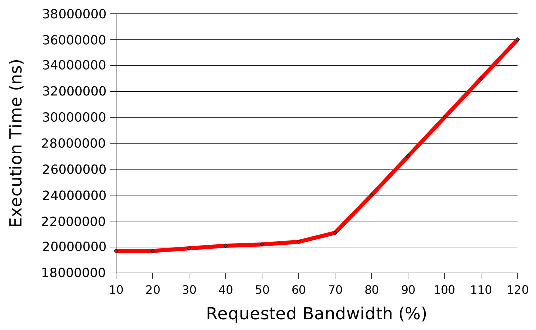


Figure 4.6: Execution time variation

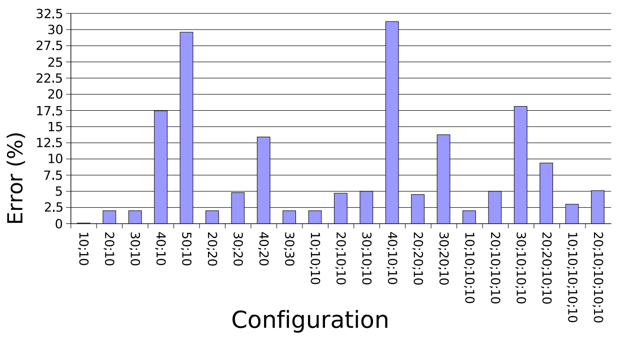


Figure 4.7: Bus additive model for different ratios of bandwidth requirements among competing tasks for bus access

However, the threshold value also depends on the ratio of bandwidth requirements of the tasks concurrently trying to access the bus. Contrarily to Fig.4.5, where each processor consumes the same fraction of bus bandwidth, Fig.4.7 shows the deviations of offered versus required bandwidth for competing tasks with different bus bandwidth requirements. Configurations with different number of processors are explored, and numbers on the x-axis show the percentage of maximum theoretical bandwidth required by each task. It can be observed that the most significant deviations arise when one task starts draining most of the bandwidth, thus creating a strong interference with all other access patterns. The presence of such communication hotspots suggests that the maximum cumulative bandwidth requirement which still stimulates an additive behaviour of the bus is lower than the one computed before, and amounts to about 50% of the theoretical maximum bandwidth. We also tried to reproduce Fig.4.7 varying the burstiness of the generated traffic. Till now, the traffic generators have used single bus transactions to stimulate bus traffic. We then generated burst transactions of fixed length (4 beat bursts, corresponding to a cache line refill of an ARM7 processor) but with varying inter-burst periods. Results are not reported here since the measured upper thresholds for the additive model are more conservative than those obtained with single transfers. Therefore, frequent single transfers and unbalanced bus utilization frequencies of the concurrent tasks running on different processors represent the worst case scenario for the accuracy of the bus additive model.

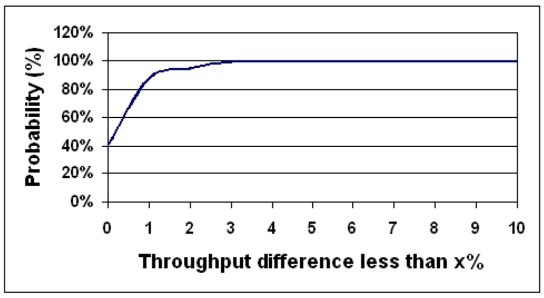


Figure 4.8: Probability of throughput differences.

4.10.2 Validation of allocation and scheduling solutions

We have deployed the virtual platform to implement the allocations and schedules generated by the optimizer, and we have measured deviations of the simulated throughput from the predicted one for 50 problem instances. A synthetic benchmark has been used for this experiment, allowing to change system and application parameters (local memory size, execution times, data size, etc.). We want to make sure that modelling approximations are not such to significantly impact the accuracy of optimizer results with respect to real-life systems. The results of the validation phase are reported in Fig.4.8, which shows the probability for throughput differences between optimizer and simulator results. The average difference between measured and predicted values is 0.76%, with 0.79 standard deviation. This confirms the high level of accuracy achieved by the developed optimization framework, thanks to the calibration of system model parameters against functional timing-accurate simulation and to the control of system working conditions.

In general, knowing the accuracy of the optimizer with respect to functional simulation is not enough, since the relative sign of the error decides whether real-time requirements will be met or not in cases where there is only very little slack time. Fig.4.9 tries to answer this question by reporting the distribution of

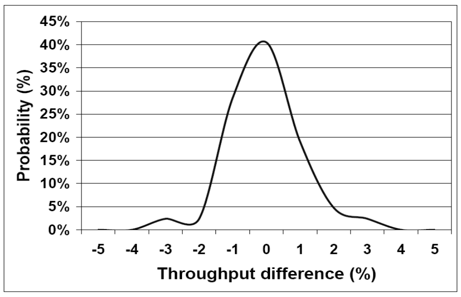


Figure 4.9: Probability of throughput differences in variable realtime study.

the sign of prediction vs measurement errors. A negative error indicates that the optimizer has been conservative, therefore the real throughput is higher than the predicted one. The contrary holds in case of positive errors. This latter case is the most critical, since it corresponds to the case where the optimizer has been optimistic. However, we clearly see that the error margin is very small (within 5%). Moreover, since the scheduling step of the optimization framework targets makespan minimization, the optimizer usually provides a schedule which results in throughput values that are far more conservative than those that were required to the optimizer. As a consequence, even if the real throughput is 5% worse, the margins with respect to the timing constraints are typically much larger.

The scalability of our approach with the number of tasks and processors has already been showed in section 4.8, and compared with state-of-the-art solving techniques. In contrast, the case studies that follow aim at proving the applicability of our approach to real-life applications and MPSoC systems. Most applications are natively coded in imperative sequential C language, and their efficient parallelization goes beyond the scope of this work. We therefore manually decomposed the GSM and MIMO benchmarks in a reasonable number of tasks and tested our mapping methodology with them.

4.10.3 Application to GSM

Most state-of-the-art cell-phone chip-sets include dual-processor architectures. GSM encoding and decoding have been among the first target applications to be mapped onto parallel multi-processor architectures. Therefore, we first proved the viability of our approach with a GSM encoder application. The source code has been parallelized into 6 pipeline stages, and each task has been pre-characterized by the virtual platform to provide parameters of task models to the optimizer. Such information, together with the results of the optimization run, are reported in Fig.4.10. Note that the optimizer makes use of 3 out of the 4 available processors, since it tries to minimize the cost of communication while meeting hardware and software constraints. The throughput required to the optimizer in this case was 1 frame/10ms, compliant with the GSM minimum requirements. The obtained throughput was 1.35 frames/ms, far more conservative. The simulation on the virtual platform provided an application throughput within 4.1% of the predicted one. The table also shows that program data has been allocated in scratchpad memory for Tasks 1,2 and 6 since they have smaller communication queues. Schedules for this problem instance are trivial. The time taken by the optimizer to come to a solution was 0.1 seconds.

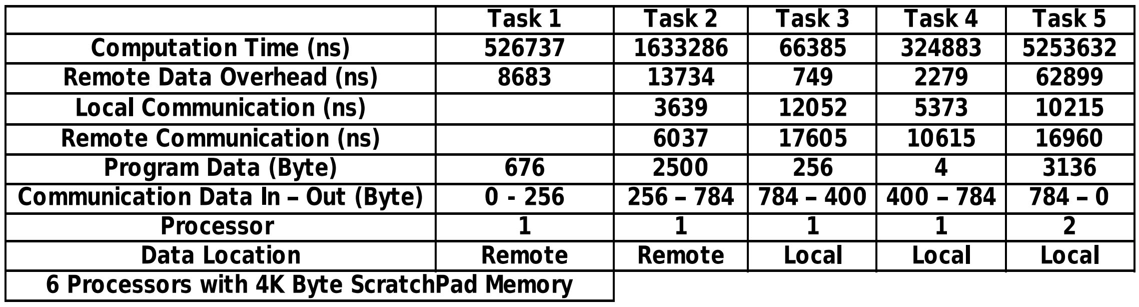
	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6
Computation Time (ns)	281639	437038	317032	308899	306213	306470
Remote Data Overhead (ns)	3978	1620	1099	2243	1916	1707
Local Communication (ns)		4754	6675	5810	6020	5810
Remote Communication (ns)		8621	12266	10773	10609	10576
Program Data (Byte)	420	420	560	560	560	560
Communication Data In – Out (Byte)	0 - 340	340 – 444	444 - 444	444 - 444	444 - 444	444 – 0
Processor	1	1	2	2	3	3
Data Location	Local	Local	Remote	Remote	Remote	Local
4 Processors with 2048 Byte ScratchPad Memory						

Figure 4.10: GSM case study.

4.10.4 MIMO processing

One major technological breakthrough that will make an increase in data rate possible in wireless communication is the use of multiple antennas at the transmitters and receivers (Multiple-input Multiple-output systems). MIMO technology is expected to be a cornerstone of many next-generation wireless communication systems. The scalable computation power provided by MPSoCs is progressively making the implementation of MIMO systems and associated signal processing algorithms feasible, therefore we applied our optimization framework to spatial multiplexing-based MIMO processing[39].

The MIMO computation kernel was partitioned into 5 pipeline stages. Optimal allocation and scheduling results for a system of 6 ARM7 processors are reported in Fig.4.11. The reported mapping configuration is referred to the case where the tightest feasible real-time constraint was applied to the system (about 1.26Mbit/sec). Obviously, further improvements of the throughput can be obtained by replacing the ARM7 cores with more computation-efficient processor cores. In this benchmark, Task 5 has the heaviest computation requirements, and requires a large amount of program data for its computation. In order to meet the timing requirements and to be able to allocate program data locally, this task has been allocated on a separate processor.



	Task 1	Task 2	Task 3	Task 4	Task 5
Computation Time (ns)	526737	1633286	66385	324883	5253632
Remote Data Overhead (ns)	8683	13734	749	2279	62899
Local Communication (ns)		3639	12052	5373	10215
Remote Communication (ns)		6037	17605	10615	16960
Program Data (Byte)	676	2500	256	4	3136
Communication Data In – Out (Byte)	0 - 256	256 – 784	784 – 400	400 – 784	784 – 0
Processor	1	1	1	1	2
Data Location	Remote	Remote	Local	Local	Local
6 Processors with 4K Byte ScratchPad Memory					

Figure 4.11: MIMO processing results.

As can be observed, the optimizer has not mapped each remaining task on a different processor, since this would have been a waste of resources providing sub-optimal results. In other words, the throughput would have been guaranteed just at the same, but at a higher communication cost. Instead, Tasks 1-4 have been mapped to the same processor. Interestingly, the sum of the local memory requirements related to communication queues leaves a very small remaining space in scratchpad memory, which allows the optimizer to map locally only the small program data of Tasks 3 and 4. The overall mapping solution was therefore not trivial to devise without the support of the combined CP-IP solver, which provides the optimal allocation and scheduling in about 600 ms. The derived configuration was then simulated onto the virtual platform, and throughput accuracy was found to be (conservatively) within 1%.

4.11 Conclusions

We target allocation and scheduling of pipelined stream-oriented applications on top of distributed memory architectures with messaging support. We tackle the complexity of the problem by means of decomposition and no-good generation, and prove the increased computational efficiency of this approach with respect to traditional ones. Moreover, we deploy a virtual platform to validate the results of the optimization framework and to check modelling assumptions, showing a very high level of accuracy. Finally, we show the viability of our approach by means of 2 demonstrators: GSM and MIMO processing. Our methodology contributes to the advance in the field of software optimization tools for highly integrated on-chip multiprocessors, and can be applied to all pipelined applications with design-time predictable workloads. The extension to generic task graphs does not present theoretical hindrances and is ongoing work.

Appendix 1: Proof of schedule periodicity

In this section we prove that despite our algorithm considers an unbounded number j of iterations of a pipeline with n tasks $Task_{ij}$, $i = 1..n$, our final schedule is always periodic. The proof assumes single token communication queues (i.e. length one queues), but it can be easily extended to any finite length.

Tasks are partitioned by the allocation module on p processors. So let us consider p partitions: $Task_{ij}\ i \in Sp_k \forall j$ where $k = 1..p$ and Sp_k is the set of tasks assigned to processor k. Our aim is to show that our (time discrete) scheduling algorithm that minimizes the makespan produces a periodic solution even if we have a (theoretical) infinite number of pipeline iterations.

The proof is based on the following idea: if we identify in the solution a state of the system that assumes a finite number of configurations, than the solution is periodic. In fact, after a given state S the algorithm performs optimal choices; as soon as we encounter S again, the same choices are performed.

For each iteration j, the state we consider is the following: the slack of each task in S_k to its deadline. The state of the system is the following: For each processor $k = 1..p$ we have $\langle Slack^k_{1j}, \ldots, Slack^k_{lj} \rangle$, where $Slack^k_{ij}$ is the difference between the deadline of $Task_{ij}$ running on processor k and its completion time. Therefore, if we prove that the number of possible state configurations is finite (i.e., it does not depend on the iteration number j), being the transitions between two states deterministic, even if we have an infinite number of repetition of the pipeline, the solution is periodic.

After the pipeline starts up, the deadline of each task $Task_{ij}$ is defined by the first iteration of task i. i.e., $Task_{i1}$. In fact, the real-time (throughput) constraint states that every P time points each task should be repeated. Therefore, if the first iteration of a task i is performed at time t_i, the second iteration of i should be performed at time $t_i + P$, and the j-th iteration at time $t_i + (j-1) * P - diration(Task_{ij})$.

Now, let us consider two cases:

- if the tasks in S_k are consecutive in the pipeline, then their repetition cannot change. For example, if tasks T_{1j}, T_{2j} and T_{3j} are allocated to the same processor (for all j), having length one queues, they can be repeated only in this order. Indeed, one can repeat T_{1j} after T_{2j}, but minimizing the makespan it is not the right decision.

- if instead the tasks in S_k are not consecutive, then there could be repetitions in between that could break the periodicity. Therefore, we should concentrate on this case.

For the sake of readibility we now omit the index representing the iteration since we concentrate on the maximum slack a task can assume. Let us consider two non consecutive tasks $T_A \in S_k$ and $T_B \in S_k$. Suppose that between T_A and T_B there are m tasks allocated on other processors different from k. Let us call them $T_{A1}, T_{A2}, \ldots T_{Am}$ ordered by precedence constraints. If we have communication queues of length one, between T_A and T_B there are AT MOST m iterations of T_A. In fact, T_A can be repeated as soon as T_{A1} starts on another processor. Also, it can be repeated as soon as another iteration of T_{A1} starts, that can happen as soon as T_{A2} starts and so on. Clearly, m iterations are possible only if

$$m * duration(T_A) \leq \sum_{i=1}^{m} duration(T_{Ai})$$

but if this relation does not hold, there can be only less iterations of T_A. Therefore, m is an upper bound on the number of iterations of T_A between the first T_A and T_B. If t_A is the time where the first repetition of T_A is performed, the m^{th} iteration of T_A has a deadline of $t_A + (m-1) * P$. Its slack is clearly bounded to the maximum deadline minus its duration, $t_A + (m-1) * P - duration(T_A)$.

The upper bound for m is $n-2$. In fact, in a pipeline of n tasks the maximum number of repetitions of a task happen if only the first and the last task are allocated on the same processor. They have $n-2$ tasks in between allocated on different processors. Therefore, the maximum number of repetitions of T_1 between T_1 and T_n is $n-2$

Therefore if the first iteration of T_1 is executed at time t_1 its $(n-2)^{th}$ iteration has a max deadline $t_1 + (n-3) * P - duration(T_1)$.

Being the max deadline of a task finite, also its max slack is finite despite the number of iteration of the pipeline.

Therefore, whatever the state is, each task belonging to the state has a finite slack. The combination of slacks are finite, and therefore, after a finite number of repetition, the system finds a state already found and becomes periodic.

Bibliography

[1] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon, "Analyzing on-chip communication in a MPSoC environment", Proc. DATE 2004.

[2] S. Prakash and A. Parker, "SOS: Synthesis of Application-Specific Heterogeneous Multiprocessor Systems", Journal of Parallel and Distributed Computing, pp. 338-351, 1992.

[3] C. Lee, M. Potkonjak and W. Wolf, "System-Level Synthesis of Application-Specific Systems Using A* Search and Generalized Force-Directed Heuristics", Procs. of the 9th Intern. Symposium on System Synthesis - ISSS '96, pp. 2-7, Nov. 1996.

[4] ARM Ltd., Sheffield, U.K., AMBA 2.0 Specification. http://www.arm.com/armtech/AMBA

[5] A. Bender, "MILP based Task Mapping for Heterogeneous Multiprocessor Systems", EURO-DAC '96/EURO-VHDL '96: Procs. of the conference on European design automation, pp. 190-197, Sept. 1996.

[6] Y. Li and W. H. Wolf, "Hardware/Software Co-Synthesis with Memory Hierarchies", 1999 IEEE Trans.on Computer-Aided Design of Integrated Circuits and Systems, pp. 1405-1417, 1999.

[7] G.De.Micheli, "Synthesis and optimization of digital circuits", McGraw Hill, 1994.

[8] B.Flachs et al., "A streaming processor unit for the CELL processor", pp.134-135, ISSCC 2005.

[9] K.S.Chatha and R.Vemuri, "Hardware-Software Partitioning and Pipelined Scheduling of Transformative Applications", 2002 IEEE Trans. on Very Large Scale Integration Systems, pp. 193-208, 2002.

[10] P.Palazzari and L.Baldini and M.Coli, "Synthesis of Pipelined Systems for the Contemporaneous Execution of Periodic and Aperiodic Tasks with

Hard Real-Time Constraints", 18th International Parallel and Distributed Processing Symposium - IPDPS'04, pp. 121-128, Apr. 2004.

[11] K.Kuchcinski and R.Szymanek, "A constructive algorithm for memory-aware task assignment and scheduling", Procs of the Ninth International Symposium on Hardware/Software Codesign - CODES 2001, pp. 147-152, Apr. 2001.

[12] G.Fohler and K.Ramamritham, "Static Scheduling of Pipelined Periodic Tasks in Distributed Real-Time Systems", Procs. of the 9th EUROMICRO Workshop on Real-Time Systems - EUROMICRO-RTS '97, pp. 128-135, June 1997.

[13] J.Axelsson, "Architecture Synthesis and Partitioning of Real-Time Synthesis: a Comparison of 3 Heuristic Search Strategies", Procs. of the 5th Intern. Workshop on Hardware/Software Codesign (CODES/CASHE97), pp. 161-166, Mar. 1997.

[14] P.Eles, Z.Peng, K.Kuchcinski and A.Doboli, "System Level Hardware/Software Partitioning Based on Simulated Annealing and Tabu search", Design Automation for Embedded Systems, pp. 5-32, 1997.

[15] S.Kodase, S.Wang, Z.Gu and K.Shin, "Improving Scalability of Task Allocation and Scheduling in Large Distributed Real-Time Systems Using Shared Buffers", Procs. of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2003), pp. 181-188, May 2003.

[16] P.Eles, Z.Peng, K.Kuchcinski, A.Doboli and P.Pop, "Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems", Procs. of the conference on Design, automation and test in Europe, pp. 132-139, 1998.

[17] K.Kuchcinski, "Embedded System Synthesis by Timing Constraint Solving", IEEE Transactions on CAD, pp. 537-551, 1994.

[18] E.S.Thorsteinsson, "A hybrid framework integrating mixed integer programming and constraint programming", Procs. of the 7th International Conference on Principles and Practice of Constraint Programming - CP 2001, pp. 16-30, 2001.

[19] A.Eremin and M.Wallace, "Hybrid Benders Decomposition Algorithms in Constraint Logic Programming", Procs. of the 7th Intern. Conference on Principles and Practice of Constraint Programming - CP 2001, pp. 1-15, Nov. 2001.

[20] I.E.Grossmann and V.Jain, "Algorithms for Hybrid MILP/CP Models for a Class of Optimization Problems", INFORMS Journal on Computing, pp. 258-276, 2001.

[21] J.N.Hooker, "A Hybrid Method for Planning and Scheduling", Procs. of the 10th Intern. Conference on Principles and Practice of Constraint Programming - CP 2004, pp. 305-316, Sept. 2004.

[22] F.Poletti, A.Poggiali and P.Marchal, "Flexible Hardware/Software Support for Message Passing on a Distributed Shared Memory Architecture", Design And Test Europe Conference 2005 Proceedings, pp. 736-741, 2005.

[23] M.Ruggiero, F.Angiolini, F.Poletti, D.Bertozzi, L.Benini, R.Zafalon, "Scalability Analysis of Evolving SoC Interconnect Protocols", Int. Symposium on System-on-Chip, 2004.

[24] J.F.Benders, "Partitioning procedures for solving mixed-variables programming problems", Numerische Mathematik, pp. 238-252, 1962.

[25] F.Poletti, P.Marchal, D.Atienza, L.Benini, F.Catthoor, J.M. Mendias, "An Integrated Hardware/Software Approach For Run-Time Scratchpad Management", DAC 2004, pp.238-243, June 2004.

[26] F.Angiolini, L.Benini, D.Bertozzi, M.Loghi and R.Zafalon, "Analyzing on-chip communication in a MPSoC environment", In Proceedings of the IEEE Design and Test in Europe Conference (DATE), pp. 752-757, Febr. 2004.

[27] J.N.Hooker and G.Ottosson, "Logic-based Benders decomposition", Mathematical Programming, pp. 33-60, 2003.

[28] RTEMS Home Page, http://www.rtems.com

[29] ARM11 MPCore,
http://www.arm.com/products/CPUs/ARM11MPCoreMultiprocessor.html

[30] CT3600 Family of Multi-core DSPs,
http://www.cradle.com/products/sil_3600_family.shtml

[31] Avispa-CH1 Communications Signal Processor,
http://www.silicon-hive.com/t.php?assetname=text&id=131

[32] P. Baptiste, C. Le Pape, W. Nuijten, "Constraint-Based Scheduling", Springer Verlag, International Series in Operations Research and Management Science, Vol. 39, 2001.

[33] Embedded microelectronic systems: status and trends, IEE Proceedings - Computers and Digital Techniques – March 2005 – Volume 152, Issue 2.

[34] S. Prakash and A. C. Parker, ”Synthesis of application-specific multiprocessor systems including memory components”, in Proceedings of the International Conference on Application Specific Array Processors, 1992.

[35] W.Wolf, “A decade of hardware/software codesign”, IEEE Computer 2003, pp.38-43, April 2003.

[36] J.C. Régin, “A filtering algorithm for constraints of difference in CSPs”, Proc. of the Twelfth National Conference on Artificial Intelligence - AAAI94, pp. 362-367, 1994.

[37] E.P.K. Tsang, “Foundation of Constraint Satisfaction”, Academic Press, 1993.

[38] Y. Kwok , I. Ahmad, ”Static scheduling algorithms for allocating directed task graphs to multiprocessors”, ACM Computing Surveys, v.31 n.4, p.406-471, 1999.

[39] D. Novo, W. Moffat, V. Derudder, B. Bougard, ”Mapping a multiple antenna SDM-OFDM receiver on the ADRES Coarse-Grained Reconfigurable Processor”, Signal Processing Systems Design and Implementation, 2005. IEEE Workshop on, 2005 Page(s):473 - 478

Chapter 5

Reducing the Abstraction and Optimality Gaps in the Allocation and Scheduling for Variable Voltage/Frequency MPSoC Platforms

5.1 Overview

Multi-core platforms are becoming widespread in a growing number of embedded application domains. These platforms require effective techniques for static allocation, scheduling and voltage/frequency assignment of complex multi-task applications. We propose a novel approach to optimally solve the allocation, scheduling and discrete voltage/frequency selection problem for MPSoCs with support for low-power features, minimizing overall system energy dissipation incurred by task execution and communications, and including frequency switching overhead. We guarantee optimality for large problem instances, which were considered beyond reach for exact solvers.

Furthermore, we have fully implemented the development-time and run-time software infrastructure required to deploy the solution computed by the optimizer on real execution platforms. This enables us to validate the effectiveness as well as the accuracy of our software optimization approach. We performed extensive analysis and validation on a cycle accurate virtual platform for a number of problem instances, including two complete application demonstrators (GSM and JPEG).

5.2 Introduction

The last five years have been characterized by a paradigm shift in the design of integrated architectures: boosting clock frequencies of monolithic processor cores has clearly reached its limits [1, 2, 3, 4], and designers are turning to multicore architectures to satisfy the ever-growing computational needs of applications within a reasonable power envelope [5]. This trend is common to both general purpose and embedded computing platforms. All commercial manufacturers of high-performance processors are currently introducing multi-core architectures (AMD's Opteron, Intel's Montecito, Sun's Niagara, IBM's Cell and Power5) [6]. Embedded computing platforms have anticipated the "multicore revolution", as they are typically designed with much tighter power budgets, aiming at maximum energy efficiency.

One of the most daunting challenges to the success of Multi-Processor System-on-Chip (MPSoC) platforms consists of developing effective software optimization tools that can optimally exploit the available cores [7]. At a coarse granularity, mapping a multi-task application to a multi-core architecture is a key step of the software development flow, as it significantly impacts design quality metrics like execution time, throughput and power. Moreover, the availability of an increasing number of architectural tuning knobs for optimal system configuration (e.g., voltage/frequency settings of processor cores, data partitioning among memory devices and/or hierarchies, synchronization techniques) is making the design space exceedingly complex for traditional modelling and solving frameworks [8]. In addition, the intricacies of component interactions in multicore architectures call for detailed system models and for their validation on a real or virtual platform [9].

In this chapter, we focus on an allocation and scheduling problem of growing practical relevance, namely finding an energy optimal mapping of a task graph onto a multi-core platform that supports independent frequency and voltage settings for all of its cores. A number of MPSoC platforms support variable frequency and voltage operation [10, 53, 12], and many authors have pointed out that optimal allocations, schedules and frequency/voltage settings lead to major power savings [13]. Unfortunately, this optimization problem is known to be NP-hard [13] even in much simplified variants [14], and most authors propose simplified models and heuristic approaches to solve it in reasonable time.

Model simplification is often achieved by abstracting away platform implementation "details" such as the penalties for frequency and voltage switching, the limited capacity of level-one scratchpad memories or the actual connectivity of communicating cores. As a result, optimization problems become more tractable, even reaching polynomial time complexity [13]. Unfortunately, this approach creates an *abstraction gap* between the optimization model and the real HW-SW platform. Validation is therefore required and the accuracy of the solutions must be carefully assessed through detailed simulation runs or execution on the target hardware.

Heuristic approaches rely on two main strategies: *problem decomposition* and *incomplete search*. Decomposition splits the problem into a set of sub-problems that are then solved in sequence. A common decomposition strategy for the mapping problem is to first perform allocation, followed by scheduling and finally voltage and frequency assignment [15, 16]. Incomplete search [17] relies on flexible iterative algorithmic frameworks (e.g., genetic algorithm or tabu search) that are customized for the target problem and generally find good solutions in a reasonable computation time. The main issue with decomposition and incomplete search is that they introduce an *optimality gap* of unknown size. In other words, they provide very limited or no information on the distance between the best computed solution and the optimal one. Even worse, when attempting to solve constrained problems, they may fail to find existing feasible solutions.

The goal of our work is to address both abstraction and optimality gaps. Namely, we formulated an accurate model for allocation, scheduling and frequency/voltage setting, which accounts for a number of ”non-idealities” in real-life hardware platforms. We also developed a novel mapping algorithm that deterministically finds optimal solutions. Even though its worst-case runtime is obviously exponential, our search strategy is computationally efficient in practice and achieves low run times (i.e. minutes) for problem instances of practical relevance (i.e. up to hundreds of tasks). This is much beyond the instance sizes that could be handled in the past by complete search algorithms, while being comparable with heuristic approaches. On the other hand, we achieve consistently lower-power results than the best previously reported heuristics. More importantly, we find feasible solutions for tightly constrained problem instances where heuristic search fails.

Furthermore, a second main contribution of our work is the implementation of the static (design-time) and dynamic (run-time) software infrastructure required to deploy the applications on the target platform. This is a critical and non-trivial task, as we must guarantee that actual execution accurately matches in time and space the solution computed by the optimizer. We can therefore validate the optimizer against cycle accurate performance and power analysis on a virtual platform. Experiments on a large number of problem instances demonstrate that the accuracy of our model is high and that execution traces match the estimates of the models with average error below 5% (worst case 10%).

Our optimizer is based on an algorithmic framework called logic-based Benders Decomposition (LBBD) [7, 19] which solves the allocation, scheduling and voltage/frequency selection problems to optimality in a computation-efficient fashion through the cooperation between two solvers: an integer linear programming (ILP) solver for allocation and voltage/frequency setting and a constraint programming (CP) solver for scheduling. It is important to emphasize that LBBD is *not* a heuristic decomposition strategy: the two solvers interact in an iterative fashion that is guaranteed to achieve convergence to optimality. The computational efficiency of our optimizer comes from three main factors: (i) we use solvers that are well matched to the sub-problems they handle (namely: ILP works well on allocation, while CP works well on scheduling), (ii) we use problem-specific strategies to propagate information from one solver to the other that rapidly achieve convergence, (iii) we use a symmetry-reducing strategy to eliminate from the search space a large number of equivalent solutions.

We target statically configured systems, where allocation, scheduling and frequency settings are precomputed at design time. Such systems require design-time predictable application behavior with small run-time fluctuations, such as many signal processing and even some multimedia applications. For these applications, our methodology makes the pre-computation of optimal solutions still affordable in spite of the increasing number of integrated processor cores and of the growing exposition of task-level parallelism.

This chapter is structured as follows. We first describe previous work in the field. The target architecture and the virtual platform environment are presented in Section 6.4. Section 5.5 provides background on optimization techniques. Our approach to the mapping problem is presented in Sections 5.6 and 7.5.2. Computation efficiency of our approach is assessed in Section 5.9. The design-time and run-time support to make the optimization framework interact with the HW-SW platform is illustrated in Sections 5.10 and 5.11. Experimental results validating the accuracy of the proposed optimizer follow in Section 5.12, while a comparison with a heuristic approach is reported in Section 5.13.

5.3 Related Work

In the following, we focus on off-line voltage/frequency selection techniques, since our approach falls into this category.

A number of techniques have been developed for single processor systems. Yao et al. proposed in [20] the first DVS approach which can dynamically change the supply voltage over a continuous range. Ishihara and Yasuura [21] modeled the discrete voltage selection problem using an integer linear programming (ILP) formulation. Xie et al. [22] present an algorithm for calculating the bounds on the power savings achievable through voltage selection. Jejurikar and Gupta [23] propose an algorithm that combines voltage scaling and shutdown in order to minimize dynamic and leakage energy in single.

Andrei et al. [13] proposed an approach that solves optimally the voltage scaling problem for multi-processor systems with imposed time constraints. The continuous voltage scaling is solved using convex nonlinear programming with polynomial time complexity, while the discrete problem is proved strongly NP hard and is formulated as mixed integer linear programming (MILP).

The previously mentioned approaches assume that the mapping and scheduling are given. However, the achievable energy savings of dynamic voltage scaling are greatly affected by the mapping and the scheduling of the tasks on the target processors.

Task mapping and scheduling are known NP complete problems [24] that have been previously addessed, without and with the objective of minimizing the energy. Both heuristic [25], [26] and exact solutions [27] have been proposed.

Assuming the mapping of the tasks on the processors is given as input, the authors from [28] present a scheduling technique that maximizes the available slack, which is then used to reduce the energy via voltage scaling. Schmitz et al. [25] present a heuristic approach for mapping, scheduling and voltage scaling on multiprocessor architectures.

A leakage-aware approach for combined dynamic voltage selection and adaptive body-biasing has been proposed in [29, 13]. However, the approach in [29] is restricted to the single processor case. A multiprocessor setting is addressed in [13] through a mixed integer linear programming approach. Although we concentrate in this chapter on the dynamic power and supply voltage selection, our methodology can handle with minor changes the combined supply and body bias scaling problem with only marginal implications on computational complexity.

The closest approach to our work is the one of Leung et al., [30]. They propose a mixed integer programming formulation for mapping, scheduling and voltage scaling of a given task graph to a target multiprocessor platform. They assume continuous voltages, so the overall result is suboptimal.

Summing up, the two main approaches followed by the system design community when facing software mapping problems in MPSoCs are: (1) either modelling and solving the problem to optimality as an Integer Program whatever the problem structure is or (2) using a special purpose heuristic algorithm requiring sophisticated debugging and tuning and achieving sub-optimal solutions. In this chapter, we claim that:

- whenever allocation and scheduling can be performed off-line due to the intrinsic features of the application (predictable workload), the correct approach is to solve these problems to optimality, thus achieving optimal system configuration performance- and energy-wise. Clearly, a computation efficient solving engine is required even for the off-line analysis, due to the complexity of the design space.
- Computation efficiency of solving techniques can be improved by analyzing and exploiting the problem structure. Integer Programming is an effective solving framework, but it is not always the best technique one can use. In general, the best solution strategy can be applied to each subproblem structure, and be deployed within a cooperative solving framework.

5.4 Target Architecture

The objective of this work is to map an application with exposed task-level parallelism onto a homogeneous multi-core platform. The main objective function consists of minimizing overall system power while meeting application real-time constraints. The degrees of freedom available for the optimization process are the allocation of tasks to processors, their scheduling in time and the frequency/voltage selection for task execution.

The target architecture for our mapping strategy is a general template for a parallel MPSoC architecture. The platform consists of a scalable number of homogeneous processing cores, a shared communication infrastructure and a shared memory for inter-tile communication. Processing cores embed instruction and data caches and are directly connected to tightly coupled software-controlled scratch-pad memories.

The architecture is assumed to provide a harmonized hardware-software support for messaging, targeting scalability to a large number of communicating cores. Messages can be exchanged by tasks through software communication queues, which can be physically allocated either in scratch-pad memory or in shared memory, depending on whether tasks are mapped onto the same processor or not. This assumption avoids to generate bus traffic and to incur congestion delays for local communications. We also target architectures where synchronization between producer-consumer pairs does not give rise to

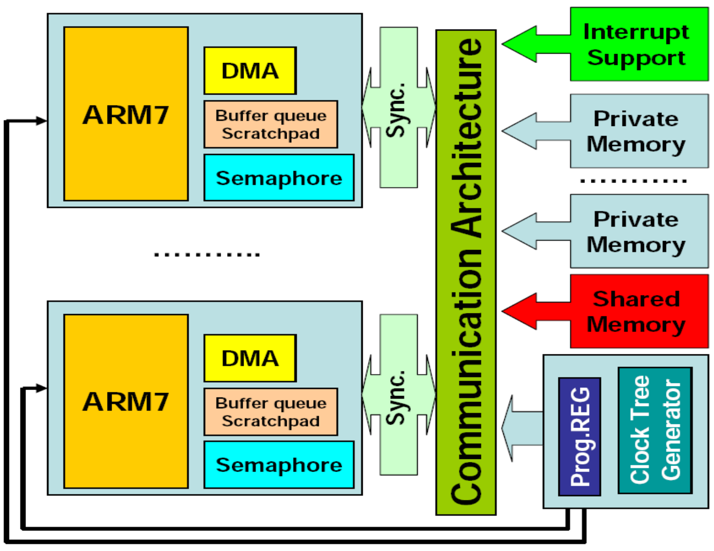


Figure 5.1: Distributed MPSoC architecture.

semaphore polling traffic on the bus, since this might unacceptably and unpredictably degrade performance of ongoing message exchanges. Interrupt-based synchronization or the implementation of distributed semaphores at each computation tile are two example mechanisms matching our requirements.

As in many recent multi-core architectures, we assume that the target platform supports different working frequencies and voltages for each processor core. In practice, each computation tile has its own clock tree, and synchronization mechanisms are provided for interfacing with the system bus (clock domain crossing). Moreover, we assume that the voltage/frequency settings can be adjusted at run-time.

An embodiment of this template architecture is considered in this work, in order to be able to provide input data to the optimization framework, to valide its solutions based on functional simulation and to validate objective function values. The architecture is illustrated in Fig.5.1. However, alternative architectures matching the same template can be input to our methodology, with just the burden to re-characterize the costs for basic communication and synchronization mechanisms, as will be explained in Section 5.12.

We used the MPARM platform for complete MPSoC functional simulation with clock-cycle accuracy in SystemC [26]. ARM7 processor cores build up the computation section of the platform, while an interconnect compliant with AMBA AHB specification is selected. 32kB instruction and data caches are instantiated. Frequency/voltage decoupling between processor cores and the bus is implemented through dual-clock FIFOs featuring a latency of 4 clock cycles of the slowest clock frequency [31]. A maximum operating frequency of 200 MHz is assumed for the bus, to which per-core frequency dividers are applied. The used voltage/frequency pairs are illustrated in Table 5.1. The variable frequency support for the platform is further enhanced by a variable clock tree generator and by programmable registers. The clock tree generator feeds the hardware modules of the platform with independent and frequency scaled clock trees. A set of programmable registers is connected to the system bus: each one of these registers contains the integer divider of the baseline frequency for each processor.

Frequency	Supply Voltage
200 MHz	1V
100 MHz	0.61V
66 MHz	0.55V
50 MHz	0.47V
40 MHz	0.44V

Table 5.1: Voltage-Frequency pairs.

Synchronization between producer/consumer pairs is implemented by means of distributed hardware semaphores. When a producer generates a message, it locally checks an integer semaphore which contains the number of free messages in the queue. If space is available, it decrements the semaphore and starts writing the message. When the message is ready, it signals this to the consumer by incrementing the consumer pointer (this is the only bus access for the entire synchronization process). Furthermore, if the semaphore is not available, the polling task registers itself on a list of tasks waiting for that semaphore and suspends itself. Other tasks on the processor can then execute. As soon as the semaphore is released, it generates an interrupt and the corresponding interrupt routine reactivates all tasks on its waiting list.

We set up a communication and synchronization library abstracting away low level architectural details to programmers, such as memory maps or explicit management of hardware semaphores and shared memory. The details can be found in [32]. System resources are controlled by the RTEMS real-time operating system.

Our virtual platform environment provides power statistics for ARM cores, caches, on-chip memories and AMBA AHB bus by leveraging technology-homogeneous power models for a 0.13 μ m technology provided by STMicroelectronics. When all tasks mapped on a processor core are suspended, then the core enters power save mode, where the power consumption is assumed to be negligible.

5.5 Background on optimization techniques

In this section, we recall the basic concepts behind the solvers we use in this chapter, namely Constraint Programming and Integer Programming. We also describe the method we use for enabling the cooperation between the two paradigms, namely the Logic Based Benders Decomposition.

5.5.1 Constraint Programming

Constraint Programming (CP) has been recognized as a suitable modelling and solving tool to face combinatorial (optimization) problems. The CP modeling and solving activity is highly influenced by the Artificial Intelligence area on Constraint Satisfaction Problems, CSPs (see, e.g., the book by [37]). A CSP is a triple $\langle V, D, C\rangle$ where V is a set of variables $X_1, \ldots, X_n$, D is a set of finite domains $D_1, \ldots, D_n$ representing the possible values that variables can assume, and C is a set of constraints $C_1, \ldots, C_k$. Each constraint involves a set of variables $V' \subseteq V$ and defines a subset of the cartesian product of the corresponding domains containing feasible tuples of values. Therefore, constraints limit the values that variables can simultaneously assume. A solution of a CSP is an assignment of values to variables which is consistent with all problem constraints.

Constraints can be either mathematical or symbolic. Mathematical constraints have the form: $t_1\ R\ t_2$ where t_1 and t_2 are finite terms, i.e., variables, finite domain objects and usual expressions, and R is one of the constraints defined on the domain of discourse (e.g., for integers we have the usual relations: $>, \geq, <, \leq, =, \neq$). For example, if two activities i and j characterized by starting times $Start_i$ and $Start_j$ and durations d_i and d_j are linked by a precedence constraint stating that activity i should be executed before activity j, the following mathematical constraint can be imposed, $Start_i + d_i \leq Start_j$. Symbolic constraints, called also global constraints, are predicates involving finite domain variables. They are expressive and powerful constraints (which can also be defined by the user) embedding constraint-dependent filtering algorithms. A typical global constraint is the

$$\textit{alldifferent}([X_1, \ldots, X_n])$$

available in most CP solvers. Declaratively, the constraint *alldifferent*$([X_1, \ldots, X_n])$ holds iff all variables are assigned to a different value. Thus, it is declaratively equivalent to a set of $n * (n-1)/2$ binary inequality constraints. However, its compact representation allows more concise models and embeds a specialized efficient graph-based filtering algorithm [36]. Many constraints have been devised for scheduling, which is the most successful application area of Constraint Programming to date. In particular, many resource and temporal constraints have been devised so as to solve large problem instances, see [16]. As an example, let us consider the cumulative constraint used for modelling limited resource availability in scheduling problems. Its parameters are: a list of variables $[S_1, \ldots, S_n]$ representing the starting time of all activities sharing the resource, their duration $[D_1, \ldots, D_n]$, the resource consumption for each activity $[R_1, \ldots, R_n]$ and the available resource capacity C. Clearly this constraint holds if in any time step where at least one activity is running the sum of the required resource is less than or equal to the available capacity. The constraint $cumulative([S_1, \ldots, S_n], [D_1, \ldots, D_n], [R_1, \ldots, R_n], C)$ holds iff

$$\forall j \sum_{S_j \le i < S_j + D_j} R_i \le C$$

In commercial solvers like ILOG Scheduler, variants of this constraint have been implemented for unary, cumulative, renewable and consumable resources in presence of preemptive and non preemptive activities.

5.5.2 Integer Programming

Another solution technique, which is well known and widely used in the system design community is Integer Programming (IP). Integer programming is an older method, with roots that date back to the late 1950s. Integer Programming can be thought of as a restriction of Constraint Programming. In fact, Integer Programming has only two types of variables: integer variables whose domain contain non-negative integers and continuous variables whose domain contain non-negative real values. In addition, IP allows only one type of constraints: linear inequalities. Finally, the objective function must be linear in the variables. It seems that these restrictions make integer programming much narrower than constraint programming. However, many problems can still be modeled effectively, and algorithms for integer programs can find optimal solutions quickly for many applications. The solving principle of IP is based on the solution of the *linear relaxation*, allowing arbitrary sets of linear constraints to be treated as a global constraint, providing a global view of the problem. The relaxation provides a bound enabling efficient pruning of the search tree and directing search toward promising regions.

The standard form of an IP is the following: let x be the vector of variables, $x = [x_1, x_2, \ldots, x_n]$. A set of these variables I are required to take on integer values, while the remaining variables can take any real value. Each variable can have a range, represented by vectors l and u such that $l_i \le x_i \le u_i$. A *linear* *constraint* on the variables is a vector of coefficients $a = [a_1, \dots, a_n]$ and a scalar right-hand-side b. The constraint is then the requirement that

$$\sum_j a_j x_j = b$$

The "=" in the constraint can also be $\leq$ or $\geq$ (but not $<$ or $>$). The objective function is formed by a vector of coefficients $c = [c_1, c_2, \dots, c_n]$, with the objective of minimizing (or maximizing) cx. An integer program consists of a single linear objective and a set of constraints. If we create a matrix $A = [a_{ij}]$, where a_{ij} is the coefficient for variable j in the ith constraint, then an integer program can be written:

$$\text{min } cx \tag{5.1}$$

$$s.t.\ \ Ax = b \tag{5.2}$$

$$l \leq x \leq u \tag{5.3}$$

$$x_j \text{ integer for all } j \in I \tag{5.4}$$

For many applications, it is worth working within the limits of integer programming to achieve high performance.

5.5.3 Logic Based Benders Decomposition

We will show that Constraint Programming and Integer Programming solvers are used to solve parts of our problem. The technique we use in this chapter for letting the two solvers cooperate is derived from a method, known in Operations Research as Benders Decomposition [24], and refined by [7] with the name of Logic-based Benders Decomposition. The classical Benders Decomposition method decomposes a problem into two loosely connected subproblems. It enumerates values for the connecting variables. For each set of enumerated values, it solves the subproblem that results from fixing the connecting variables to these values. The solution of the subproblem generates a constraint, called Benders cut, that the connecting variables must satisfy in all subsequent solutions enumerated. The process continues until the master problem and subproblem converge providing the same value. The classical Benders approach, however, requires that the subproblem be a continuous linear or nonlinear programming problem. This requirement poses severe applicability restrictions. For instance scheduling is a combinatorial problem that has no practical linear or nonlinear programming model. Therefore, the Benders decomposition idea can be extended to a logic-based form (Logic Based Benders Decomposition - LBBD) that accommodates an arbitrary subproblem, such as a discrete scheduling problem. More formally, as introduced in [7], a problem can be written as

$$\text{min } f(x, y) \tag{5.5}$$

$$s.t.\ p_i(y)\ i \in I_1 \ \text{ Master Problem Constraints} \tag{5.6}$$

$$g_i(x)\ i \in I_2 \ \text{ Subproblem Constraints} \tag{5.7}$$

$$q_i(y) \to h_i(x)\ i \in I_3 \ \text{Conditional Constraints} \tag{5.8}$$

$$y \in Y \ \text{ Master Problem Variables} \tag{5.9}$$

$$x_j \in D_i \ \text{ Subproblem Variables} \tag{5.10}$$

We have master problem constraints, subproblem constraints and conditional constraints linking the two models. If we solve the master problem to optimality, we obtain values for variables y in I_1, namely $\bar{y}$, and the subproblem is thus formulated as

$$\text{min } f(x, \bar{y}) \tag{5.11}$$

$$g_i(x)\ i \in I_2 \ \text{ Subproblem Constraints} \tag{5.12}$$

$$q_i(\bar{y}) \to h_i(x)\ i \in I_3 \ \text{ Conditional Constraints} \tag{5.13}$$

$$x_j \in D_i \ \text{ Subproblem Variables} \tag{5.14}$$

The heart of Benders decomposition is somehow to derive a function that gives a valid lower bound on the optimal value of the original problem for any fixed value of y. This function yields to a valid Benders cut. The algorithm proceeds as follows. At each iteration $1..h$ the Benders cuts so far generated are added to the master problem model that becomes

$$\min f(x,y) \tag{5.15}$$

$$s.t.\ p_i(y)\ i \in I_1 \ \text{ Master Problem Constraints} \tag{5.16}$$

$$B_{y_i}(y)\ i \in 1..h \ \text{ Benders cuts} \tag{5.17}$$

$$y \in Y \ \text{ Master Problem Variables} \tag{5.18}$$

y_i is the solution found at iteration i of the master problem.

In practice, to avoid the generation of master problem solutions that are trivially infeasible for the subproblem, it is worth adding a relaxation of the subproblem to the master problem.

Deciding to use the LBBD to solve a combinatorial optimization problem implies a number of design choices that strongly affect the overall performance of the algorithm. Design choices are (i) how to decompose the problem, (ii) which solver to choose for each decomposition; (iii) which model to use for feeding each solver; (iv) which subproblem relaxation to add to the master problem so as to avoid the generation of trivially infeasible solutions; (v) which Benders cuts to define for establishing the interaction between the master and the sub-problem.

5.6 High-impact modeling choices

We first tried to solve the overall mapping problem (allocation and scheduling) to optimality using a single approach. We tested both Constraint Programming and Integer Programming alone on the problem with unacceptable computation efficiency results. Therefore, we switched to Logic Based Benders Decomposition. As shown in section 5.5.3, a number of design choices has to be addressed.

- **How to decompose the problem.** We split the overall mapping problem into two sub-problems:
  (1) the allocation of tasks to processors and the selection of a baseline frequency scaling factor for the execution of each task. The objective function at this stage is the minimization of the energy spent by the system for task execution and communication.
  (2) The scheduling of tasks in time on the assigned processors. The objective function of the scheduling sub-problem is the secondary objective function of the mapping problem as a whole, and consists of energy min-

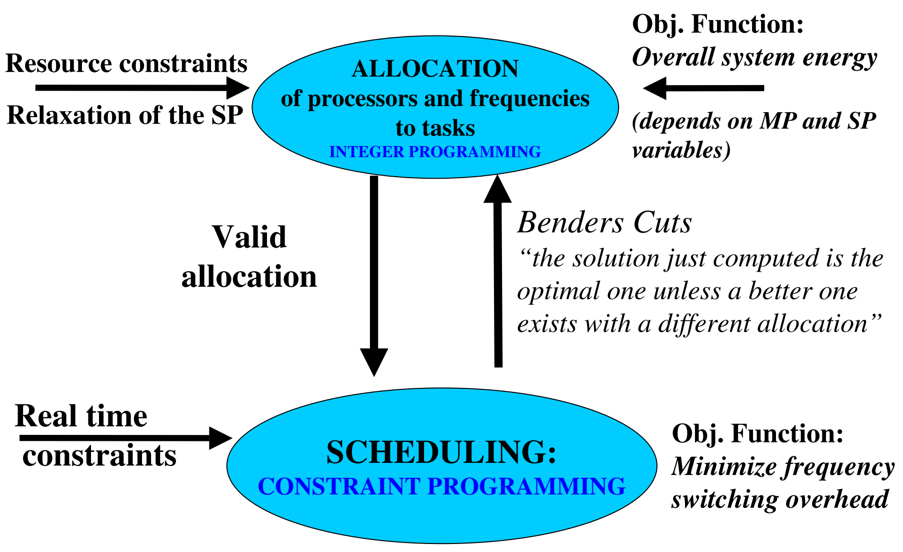


Figure 5.2: Application of Logic-Based Benders Decomposition to the Dynamic Voltage Scaling Problem.

imization associated with frequency switchings.

Interestingly, our decomposition choice and interaction strategy can potentially result in higher accuracy with respect to competing solving strategies, such as [6] [8]. These latter cope with scheduling problems where tasks assigned to different machines are not linked by any constraint. Therefore, the subproblem there is composed by a set of independent single machine scheduling problems.

Different objective functions can be easily supported by our technique. Clearly, one should change the relaxation of the subproblem and the Benders cuts. The aim of this chapter is not to prove the effectiveness of Logic-Based Benders Decomposition in general, but specifically for the problem at hand.

- **Which solver to choose for each decomposition.** There are no general guidelines for choosing the best solver for the problem at hand. For some problems, it is widely recognized that either Integer Programming or Constraint Programming are the techniques of choice. Integer Programming is effective for coping with optimization problems, it has a global problem view due to the use of linear relaxations, but sometimes its models are too large and somewhat unnatural. On the other hand, Constraint Programming has an effective way to cope with the so called *feasibility reasoning*, encapsulating efficient and incremental filtering algorithms into global constraints. However, CP has a naive way to cope with optimization problems by successively solving a set of constraint satisfaction problems with tighter bounds on the objective function.

For the problem at hand, the allocation problem has been solved via Integer Programming. It better copes with objective functions based on the sum of assignment costs. The model feeding the IP solver will be described in section 5.7.1. For the scheduling problem, the solver is instead based on Constraint Programming since it better copes with temporal resource constraints and finer time granularities. This part will be described in detail in section 5.7.2.

- **Which relaxation to use.** Since real-time constraints are not taken into account in the allocation problem solver, this latter runs the risk of providing trivially infeasible solutions. In other words, computation and communication activities may be packed in the same processor in such a way that the sum of their durations exceeds the real-time constraints. This makes the considered assignment certainly infeasible for the overall problem. This can be avoided by adding a relaxation of the subproblem to the master problem model. In particular, we should state in the master problem that the sum of the overall execution and communication times should not exceed the deadline for that processor. It follows from this that the computed allocation will be much more similar to a feasible one for the problem at hand and this reduces the search time. In addition, using a subproblem relaxation we can compute a bound on the energy and the time for frequency switching for tasks allocated on the same processor. The relaxations used are described in section 5.7.3.

- **Which Benders cuts to use.** This aspect is essential for the interaction between the two solvers. We solve the allocation problem first (called master problem), and the scheduling problem (called subproblem) later. The master is solved to optimality and its solution passed to the subproblem solver. In contrast to 4, where only performance issues were considered and the primary objective function involved only variables of the master problem, now the overall system energy minimization function involves also sub-problem variables (the frequency switching overheads). For this reason, the iterative solving strategy is key to converge to the optimal solution. In fact, the scheduling sub-problem solver may indicate that no feasible schedule exists for a given allocation. In this case, the master problem solver will be constrained not to return the same allocation through proper Benders Cuts. Alternatively, a feasible schedule is derived for the given allocation, and a new iteration of the master problem solver is triggered. This way, the computed allocation and scheduling solution at the first iteration is retained as the optimal solution unless a more energy-efficient one exists with a different allocation and frequency assignment. More details follow in Section 5.7.4.

The resulting cooperative solving framework for the power-aware mapping problem is summarized in Fig.5.2 and detailed in Section 7.5.2.

5.7 Dynamic Voltage Scaling Problem - DVSP

We consider a task graph G whose nodes represent a set of T tasks, that are annotated with their deadline dl_t and with the worst case number of clock cycles WCN_t. Arcs represent dependencies among tasks. Each arc is annotated with the amount of data two dependent tasks t_i and t_j should exchange, and therefore the number of bus clock cycles for exchanging (reading and writing) these data $WCN_{Rt_it_j}$ and $WCN_{Wt_it_j}$. Both the read and write activities are performed at the same speed of the task and use the bus (which instead works at the maximum speed). Execution, read and write activities are modelled as atomic. Tasks run on a set of processors P. Without lack of generality, we assume that each task has enough local memory to meet its storage requirements, since these latter can be easily included in an extended model version.

Each processor can run with M energy/speed modes and has a maximum load constraint dl_p. Each task spends energy both in computing and in communicating. In addition, when the processor switches between two modes it spends time and energy. We have a matrix $\mathbf{E}$ describing energy overhead $E_{f_if_j}$ for switching from any frequency f_i to any f_j. Similarly, a matrix $\mathbf{T}$ describing time switching overhead $T_{f_if_j}$ is defined.

The Dynamic Voltage Scaling Problem (DVSP) is the problem of allocating tasks to processors, defining the running speed of each task and scheduling each of them minimizing the total energy consumed. In order to solve the DVSP to optimality without simplifying assumptions relieving computation constraints but impairing solution accuracy, we applied the Logic-Based Benders decomposition technique [7] to this new application domain.

As introduced in Section 5.6, we decompose the problem in two parts: the first, called Master Problem, is the allocation of processors and frequencies to tasks and the second, called Subproblem, is the scheduling of tasks given the static allocations and frequency assignments provided by the master. The master problem is tackled by an Integer Programming solver while the subproblem through a Constraint Programming solver.

5.7.1 The Master Problem model

We model the allocation problem using Integer Linear Programming. We have binary variables X_{ptm} which take value 1 if task t is mapped on the processor p and runs in (energy-speed) mode m, 0 otherwise. Since we also take into account communication, we assume that two tasks consume energy and time for communication only if they are allocated on two different processors. Variables $R_{pt_1t_2m}$ and $W_{pt_1t_2m}$ take value 1 if the task t_1 running on processor p reads (resp. writes) data (at mode m) from (resp. for) a task t_2 not running on p. We assume that tasks running on the same processor do not consume energy and do not spend time in communication for the sake of the optimization problem, while we include the actual minor costs for local communication in execution time and energy for the sake of modelling accuracy. They are input data provided with the task graph.

Any task can be mapped to only one processor and can run at only one speed, that is:

$$\sum_{p=1}^{P}\sum_{m=1}^{M} X_{ptm} = 1 \ \ \forall t$$

Also, each task reads data (resp. writes data) atomically while executing in a given mode and on a given processor, thus constraining variables $R_{pt_1t_2m}$ and $W_{pt_1t_2m}$:

$$\sum_{p=1}^{P}\sum_{m=1}^{M} R_{pt_1t_2m} \leq 1 \ \ \forall t_1, t_2$$

$$\sum_{p=1}^{P}\sum_{m=1}^{M} W_{pt_1t_2m} \leq 1 \ \ \forall t_1, t_2$$

Since at each write activity corresponds a related read activity, we have:

$$\sum_{p=1}^{P}\sum_{m=1}^{M} (W_{pt_1t_2m} - R_{pt_2t_1m}) = 0 \ \ \forall t_1, t_2$$

The objective function OF is to *minimize the energy consumption for task execution E_{comp} and for task communication E_{read} and E_{write}*:

$$E_{comp} = \sum_{p=1}^{P}\sum_{m=1}^{M}\sum_{t=1}^{T} X_{ptm} WCN_t t_{clock_m} P_{tm}$$

$$E_{Read} = \sum_{p=1}^{P}\sum_{m=1}^{M}\sum_{t,t1=1}^{T} R_{ptt_1m} WCN_{Rtt_1} t_{clock_m} P_{tm}$$

$$E_{Write} = \sum_{p=1}^{P} \sum_{m=1}^{M} \sum_{t,t1=1}^{T} W_{ptt_1m} WCN_{Wtt_1} t_{clock_m} P_{tm}$$

$$OF = E_{comp} + E_{Read} + E_{Write}$$

where P_{tm} is the power consumed by task t when running in execution mode m and t_{clock_m} is the clock cycle at mode m.

The objective function defined up to now depends only on master problem variables. However, switching from one speed to another introduces transition costs, but their value can be computed only at scheduling time. Therefore, we update the objective function of the master problem with frequency transition (or setup) costs:

$OF_{Master} = OF + \sum_{p=1}^{P} Setup_p$

where $Setup_p$ is the cost of frequency switching on processor p. Note that in the master problem model the $Setup_p$ variables are not constrained. This is true only in the first iteration of the Logic Based Benders Decomposition algorithm where all the $Setup_p$ variables are forced to be 0. From the second iteration on, instead, cuts are produced by the subproblem, constraining variables $Setup_p$ such that they might no longer be 0. These cuts will be described in section 5.7.4. In addition, for this variable, we can compute a bound using a relaxation of the subproblem. We will explain this relaxation in section 5.7.3.

In the master problem model, we have added a set of constraints that avoid the computation of symmetric (thus useless) solutions. For example, the first task is always allocated on the first processor. Each task i should be allocated on a processor j only if $j \leq i$. In addition a task uses a new processor only if it is not mappable on a processor already used.

5.7.2 The Sub-Problem model

For the scheduling part we use a Constraint Programming model. Each task t has an associated variable representing its starting time $Start_i$. The duration is fixed since the frequency has been decided in the master problem, i.e., $duration_i = WCN_i/f_i$. In addition, if two communicating tasks t_i and t_j are allocated on two different processors, we should introduce two additional activities (one for writing data to the shared memory and one for reading data from the shared memory). We model the starting time of these activities $StartWrite_{ij}$ and $StartRead_{ij}$. These activities are carried on at the same frequency of the corresponding task. If t_i writes and t_j reads data, the writing activity is performed at the same frequency of t_i and its duration $dWrite_{ij}$ depends on the frequency and on the amount of data t_i writes, i.e., WCN_{Wij}/f_i.

Analogously, the reading activity is performed at the same frequency of t_j and its duration $dRead_{ij}$ depends on the frequency and on the amount of data t_j reads, i.e., $WCN_{R_{ij}}/f_j$. Clearly the read and write activities are linked to the corresponding task:

$Start_i + duration_i \leq StartWrite_{ij} \;\; \forall j$

$StartRead_{ij} + dRead_{ij} \leq Start_j \;\; \forall i$

The constraint is not an equality constraint since each task can produce data for many tasks and can read from many tasks. Moreover, reads and writes on the same queue are linked from

$StartWrite_{ij} + dWrite_{ij} \leq StartRead_{ij} \;\; \forall i, j$

The way reading and writing activities are scheduled heavily depends on the the task graph structure. If we restrict our analysis to pipelined task graphs (i.e., dependencies among tasks are such that they are logically ordered in a pipeline, as in 4), then input data reading activities can be considered tightly coupled with computation activities of each task. Therefore, tasks writing their output data to shared memory just have their execution time increased by a quantity WCN_W/f_m, where WCN_W is the number of clock cycles for writing data (it depends on the amount of data to write) between a task and its successor in the pipeline, and f_m is the frequency of the clock when task t is performed. Similarly, tasks reading input data from shared memory have their duration increased by a quantity WCN_R/f_m.

On the contrary, for generic task graphs, a task might need to read multiple input queues before executing, with possible suspensions between the consecutive reading activities, as illustrated in Fig.5.3. Our modelling framework accounts for this general case.

Therefore, we introduce constraints forcing the execution of a task to start immediately after its last reading activity is completed, and the writes of one task to be executed sequentially without intermediate suspensions beginning from the execution completion of that task. For this purpose, we need to introduce two additional activities for each task named $MacroRead_i$ and $MacroWrite_i$. These latter group all the reading and writing activities of the associated task with index i. Durations of these macro-activities can be expressed as (symbol $\rightarrow$ indicates a precedence constraint):

$dMacroRead_i \geq \sum_{j,j \rightarrow i} dRead_{ij} \;\; \forall i$
$dMacroWrite_i = \sum_{j,i \rightarrow j} dWrite_{ij} \;\; \forall i$

This leads to new constraints linking communication and execution activities:

$Start_i + duration_i = StartMacroWrite_i \;\; \forall i$

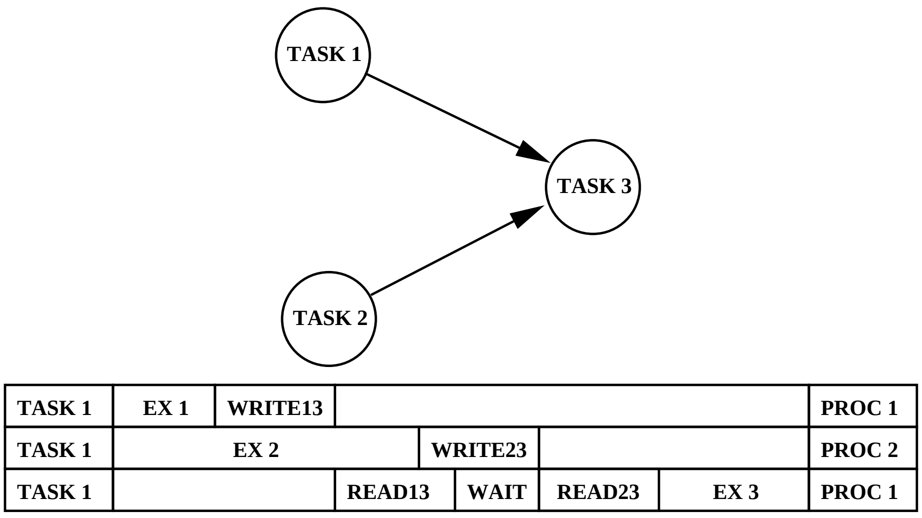


Figure 5.3: Example of multiple input data reads and their scheduling in time.

$StartMacroRead_i + dMacroRead_i = Start_i \;\; \forall i$

In the subproblem, we model precedence constraints in the following way: if tasks t_i should precede task t_j and they run on the same processor at the same frequency the precedence constraint is simply:

$Start_i + Duration_i \leq Start_j$

If instead the two tasks run on the same processor at different speed, we should add the time $T_{f_i f_j}$ for switching between the two frequencies.

$Start_i + Duration_i + T_{f_i f_j} \leq Start_j$

If the two tasks run on different processors and should communicate we should add the time for communicating.

$Start_i + Duration_i + dWrite_{ij} + dRead_{ij} \leq Start_j$

The scheduling engine must also verify that timing and resource requirements are met. As regards timing, task as well as processor deadlines are forced with proper constraints. In the simplifying assumption that task and processor deadlines are set to the same value, we just have to check that

$Start_i + Duration_i \leq Deadline \;\; \forall i$

otherwise the generalization is straightforward.

Resources are modelled as follows. We have a unary resource constraint for each processor, modelled through a cumulative constraint having as parameters a list of all tasks sharing the same resource p, $TaskList_p$, their durations $DurationList_p$, their resource consumption (which is a list of 1) and the capacity of the processor which is 1

$cumulative(TaskList_p, DurationList_p, [1], 1) \;\; \forall p$

We model the bus through an additive model we have already validated in 4. Based on this model, the bus can be shared by many activities (reads/writes from/to shared memory) in such a way that the offered bandwidth equals the sum of the bandwidth requirements of the concurrent activities. In principle, this is true only provided congestion effects remain marginal. In practice, when the bandwidth requirements exceed a given upper threshold, the offered bandwidth incurs a saturation effect, since most of the time is spent by communication actors in competing for bus access. Operating in this regime might not be convenient, since interconnect performance becomes unpredictable (e.g., the additive model fails) and only modest gains in interconnect effective utilization are achieved at the cost of an impairment of modelling accuracy.

The objective function we want to minimize in the scheduling problem is the setup energy, i.e., the energy spent for frequency switchings:

$min \sum_{p=1}^{P} Setup_p$

For this purpose, we use a matrix of precomputed transition costs $\mathbf{E}$ which reports the energy overhead for switching from frequency f_i (row i) to f_j (column j), and whose diagonal is obviously null. If we indicate with S_p the set of task pairs which are scheduled consecutively on processor p, then the setup costs can be derived as

$Setup_p = \sum_{(i,j)\in S_p} E_{f_i f_j} \;\; \forall p$

A bound on $Setup_p$ is computed in Subsection 5.7.3.

5.7.3 Relaxation of the subproblem

The master problem formulation described in section 5.7.1 will result in allocations where tasks will potentially run at their lowest frequencies and on the same processor, since task and processor deadlines are not yet accounted for in the master problem. Feeding these allocation solutions to the subproblem solver will most probably result in infeasible schedules, thus leading to a lot of computation-inefficient iterations between master and sub-problem. To avoid this, we introduce relaxations of the subproblem in the master problem model. In other words, we impose that on each processor the sum of the time spent for the computation, plus the time spent for communication (read and write) should be less than or equal to the deadline of the processor, in order to prevent trivially infeasible solutions:

$$T^p_{comp_p} = \sum_{t=1}^{T}\sum_{m=1}^{M} X_{ptm}\frac{WCN_t}{f_m}$$

$$T^p_{read} = \sum_{t=1}^{T}\sum_{m=1}^{M}\sum_{t=1}^{T} R_{ptt_1m}\frac{WCN_{Rtt_1}}{f_m}$$

$$T^p_{write} = \sum_{t=1}^{T} \sum_{m=1}^{M} \sum_{t=1}^{T} W_{ptt_1m} \frac{WCN_{Wtt_1}}{f_m}$$

$$T^p_{comp} + T^p_{read} + T^p_{write} \leq dl_p \quad \forall p$$

In the same way, task deadlines can be captured, which are the same formulas but the final sums are computed for each task.

Note that to further improve these constraints, we can add a contribution concerning the setup time T_{switch}, i.e., the time spent to switch between two frequencies in the same processor.

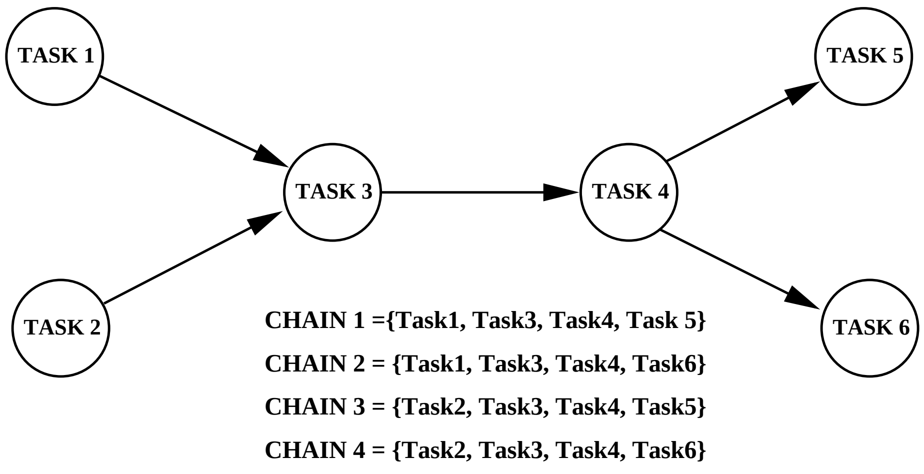
$$T^p_{comp} + T^p_{read} + T^p_{write} + T^p_{switch} \leq dl_p \quad \forall p$$

On T_{switch} we can only compute a lower bound since the real switching time can be computed once the schedule is known. The idea is the following. If we consider all the task frequencies allocated on a single processor, we know that T_{switch} is at least the sum of all switches minus the greatest switch time. For instance if frequencies f_1 f_2 and f_3 are allocated on processor $PE0$, we have to sum the minimum time for switching to frequency f_1, f_2 and f_3 minus the maximum of the three. To this purpose we have defined variables Z_{pf} taking value 1 if the frequency f is allocated at least once on the processor p, 0 otherwise. In addition we can extract from the matrix **T** of switching time overheads a vector $\bar{T}$ corresponding to the time of frequency switches that will be possibly performed on the processor. The i-th element in the vector $\bar{T}$ is the minimum time for switching to frequency i. The lower bound on T_{switch} can be imposed as follows:

$$T^p_{switch} \geq \sum_{f=1}^{M} (Z_{pf}\bar{T}_f - max_f\{\bar{T}_f | Z_{pf} = 1\}) \;\; \forall p$$

An additional set of constraints has been introduced to further restrict the search space and improve computation-efficiency. In particular, we constrain execution mode selection m for tasks belonging to a *precedence chain*, (independently of the processor they use) forcing the sum of their execution time not to exceed processor deadline of the last task $dl_{p_{last}}$. A precedence chain includes tasks featuring precedence constraints in a pipelined fashion, wherein the first and the last task in the chain do not feature neither producer nor consumer tasks, respectively. An example of the extraction of 4 precedence chains from a simple task graph is showed in Fig.5.4, and the constraint in the problem model is as follows:

$T^c_{comp} + T^c_{read} + T^c_{write} \leq dl_{p_{last}} \quad \forall c(chains)$



CHAIN 1 ={Task1, Task3, Task4, Task 5}

CHAIN 2 = {Task1, Task3, Task4, Task6}

CHAIN 3 = {Task2, Task3, Task4, Task5}

CHAIN 4 = {Task2, Task3, Task4, Task6}

Figure 5.4: Precedence chain extraction from an example task-graph.

Another aspect of the relaxation that helps in avoiding the computation of suboptimal solutions concerns the computation of a bound on the switching costs on each processor $Setup_p$. It is computed in the same way described for T_{switch}. This time, however, we have to extract from the matrix of switching cost overheads **E** a vector $\bar{E}$ corresponding to the cost of frequency switches that will be possibly performed on the processor. The i-th element in the vector $\bar{E}$ is the minimum cost for switching to frequency i.

$$Setup_p \geq \sum_{f=1}^{M} (Z_{pf}\bar{E}_f - max_f\{\bar{E}_f | Z_{pf} = 1\}) \ \forall p$$

5.7.4 Benders Cuts

Once the subproblem has been solved, we generate Benders Cuts. The cuts are of two types:
(i) if there is no feasible schedule given an allocation, we have to compute a no-good on variables X_{ptm} avoiding the same allocation to be found again.
(ii) if a feasible and optimal schedule exists, we cannot simply stop the iteration since the master objective function depends also on subproblem variables. Therefore, we have to produce cuts saying that the one just computed is the optimal solution unless a better one exists for a different allocation. These cuts produce a lower bound on the setup costs of the processors.

The procedure converges when the master problem produces a solution with the same objective function value of the previous one.

The first type of cuts are no-good: we call J_p the set of couples (Task, Frequency) allocated to processor p. We impose

$$\sum_{(t,m)\in J_p} X_{ptm} \leq |J_p| - 1 \ \forall p$$

Let us concentrate on the second type of cuts. The cuts we produce in this case are bounds on the variable $Setup$ previously defined in the Master Problem.

Suppose the schedule we find for a given allocation has an optimal setup cost $Setup^*$. It is formed by independent setups, one for each processor $Setup^* = \sum_{p=1}^{P} Setup_p^*$.

We have a bound on the setup LB_{Setup_p} on each processor and therefore a bound on the overall setup $LB_{Setup} = \sum_{p=1}^{P} LB_{Setup_p}$.

$$Setup_p \geq 0$$

$$Setup_p \geq LB_{Setup_p}$$

$$LB_{Setup_p} = Setup_p^* - Setup_p^* \sum_{(t,m)\in J_p} (1 - X_{ptm})$$

These cuts remove only one allocation. Indeed, we have also produced cuts that remove some symmetric solutions.

We have devised tighter cuts removing more solutions. However, they complicate the model too much and our experimental results show that these cuts, even if tighter, do not lead to any advantage in terms of computational time.

5.8 Example of computation

We now show a simple example of how the two solvers work and interact on a small problem instance with 5 computation tasks and 4 communication tasks, with the precedence constraints as described in Figure 5.5. Table 5.2 shows the duration (in clock cycles) of task executions. For communication tasks we report the duration in case the communication is local to the processor or remote. The durations of the reading and the writing activities of each communication Com_i are the half of the values reported in Table 5.2. We have 2 processing elements that run at 3 different frequencies, 200MHz, 100MHz and 50MHz. Therefore, for example $Task_1$ will last 1.5ms if it runs at 200MHz, 3ms if it runs at 100MHz and 6ms if it runs at 50MHz. The energy spent by the processors are 10mW when running at 200MHz, 5mW when running at 100MHz and 2mW when running at 50MHz.

Each frequency switching needs 200ns to be performed. In addition, switching from 200MHz to any other frequency leads to an energy consumption of 500pJ; switching from 100MHz to any other frequency leads to an energy consumption of 200pJ; switching from 50MHz to any other frequency leads to an energy consumption of 100pJ.

The real-time requirement imposes the processors deadline at 7ms.

Name	Task0	Task1	Task2	Task3	Task4
Clock	300k	100k	300k	100k	100k
Name	Com1	Com2	Com3	Com4	
Clock	100/200	100/200	100/200	100/200	

Table 5.2: Activity durations for the example

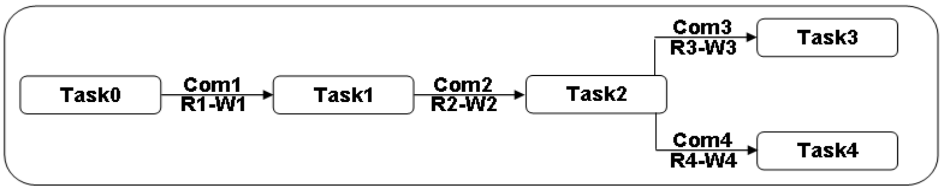


Figure 5.5: Task graph for the example in Table 5.2

The first solution found by the master problem allocates $Task_3$ alone on the first processor and all the other tasks on the second processor[1]. $Task_0$ and $Task_2$ run at the highest speed, $Task_3$ and $Task_4$ run at the lowest speed and $Task_1$ runs at 100MHz. The master problem objective function (cost for execution and communication) is 43023400 pJ, and the lower bound provided by the subproblem relaxation for the energy spent for frequency switching is 300pJ. This solution is passed to the scheduling problem solver that checks the feasibility of the solution found. In this case, the solution is feasible and the scheduler provides an optimal solution that minimizes the energy for frequency switching whose cost of 1200 pJ. The solution for the overall problem is the sum of the energy spent for execution, communication and switching, i.e., 43024600 pJ.

Now a Benders Cut is produced providing a bound on the cost. The cutting plane states that 43024600 pJ is the optimal solution, unless a better one exists for a different allocation and frequency assignment.

A new allocation is found taking into account the cutting plane just generated. The new solution has an objective function value, 43023500, worse than the first one. In this case, the process allocation is the same, but now also $Task_1$ runs at the highest speed. We have an higher power consumption for

[1]Note that an equivalent symmetric solution can be obtained just by interchanging the two processors.

execution, but we can avoid 2 frequency switchings. The lower bound on the switching overhead is in fact 100 pJ, thus this allocation can potentially lead to a better solution. The solution is passed to the subproblem solver. A scheduling is produced whose actual switching cost is 500 pJ. The overall solution, whose cost is 43024000 pJ, is the best one found so far.

Another Benders Cut is produced and passed to the master problem. The third allocation is produced whose cost is 43023500 pJ and switching lower bound of 300 pJ. This cost is lower than the overall cost of the best solution found. The subproblem schedules this allocation and finds the optimal switching overhead of 1200 pJ. The overall cost, 43024700 pJ, is actually worse than the one found at the previous iteration.

The master problem does not find any allocation with an objective function lower than 43024000 pJ and the search ends.

The optimal allocation is: $Task_3$ on the first processor and all the other tasks on the second processor. The optimal frequency assignment is: $Task_0$, $Task_1$ and $Task_2$ at the higher speed and the others at the lower. The overall optimal cost is 43024000 pJ.

5.9 Computational efficiency

We tested the computational efficiency of our hybrid approach on a 2GHz Pentium 4 machine with 512 Mb RAM and leveraged state-of-the-art professional solving tools, namely ILOG CPLEX 8.1, ILOG Solver 5.3 and ILOG Scheduler 5.3. We considered two kinds of DVSP instances: i) instances with a pipelined task graph and ii) instances with a generic task graph.

5.9.1 Pipelined task graphs

The pipeline workflow is typical, for example, of signal processing applications (e.g., baseband processing, video graphics pipelines). The same set of tasks is repeated on each input data unit or frame. We cannot know in advance the number of frames, thus, to analyze the pipeline at working rate, thus we schedule several repetition of each task. If n is the number of tasks in the pipeline, after n repetitions the pipeline is at full rate. In a pipeline with n tasks, we have n execution activities and $2 \times (n-1)$ communication activities (a read and a write for each edge in the graph); we therefore allocate $n + 2 \times (n-1)$ and schedule $n \times (n + 2 \times (n-1))$ activities.

We generated and solved 280 instances with an increasing number of tasks and processing elements. Results are summarized in Table 5.3. The first three columns contain the number of allocated and scheduled activities (execution+communication data writes and reads) and the number of processing elements considered in the instances (we consider here PEs able to run at three different frequencies). The last two columns represent respectively the search time and the number of iterations between the master and the subproblem. Each value is the mean over 10 instances with the same number of tasks and PEs. We can see that for all the instances the optimal solution can be found within four minutes and the algorithm scales quite smoothly for increasing number of tasks and PEs. We can also see that the number of iterations is typically low: the optimal solution can be found after one iteration in the 50% of the cases and the number of iterations is at most 5 in almost the 90% of cases. This result is due to the tight relaxations added to the master problem model.

Activities				
Alloc	**Sched**	**Procs**	**Time (s)**	**Iters**
4+6	16+24	2	1,73	1,98
4+6	16+24	3	1,43	2,91
4+6	16+24	4	2,24	3,47
5+8	25+40	2	2,91	2,36
5+8	25+40	3	4,19	4,12
5+8	25+40	4	5,65	4,80
5+8	25+40	5	6,69	3,41
6+10	36+60	2	3,84	2,90
6+10	36+60	3	10,76	2,17
6+10	36+60	4	15,25	4,66
6+10	36+60	5	23,17	4,50
6+10	36+60	6	26,14	3,66
7+12	49+84	2	4,67	1,75
7+12	49+84	3	5,90	1,90
7+12	49+84	7	34,53	6,34
8+14	64+112	2	4,09	3,28
8+14	64+112	3	10,99	1,83
8+14	64+112	4	12,34	4,45
8+14	64+112	5	22,65	10,53
8+14	64+112	7	51,07	6,98
9+16	81+144	2	1,79	1,12
9+16	81+144	5	60,07	7,15
9+16	81+144	6	70,40	9,20
10+18	100+180	2	5,52	1,83
10+18	100+180	3	3,07	1,96
10+18	100+180	6	120,02	6,23
10+18	100+180	10	209,35	10,65

Table 5.3: Search time and number of iterations for instances with pipelined task graphs

5.9.2 Generic task graphs

We extended our analysis to instances where the task graph is a generic one, so an activity can possibly read data from more than one preceeding activities and possibly write data that will be read by more than one subsequent activity. The number of reading and writing activities can become considerably higher, being higher the number of edges in the task graph. We consider here PEs that can run at six different frequencies. This problem is much harder than the pipelined one, because the task graph can have a number of parallel task execution chains and thus the macro-activities described in Section 5.7.2 must be considered, complicating and introducing symmetries in the model. Differently from the pipelined instances, we schedule a single repetition of each task.

Table 5.4 summarizes the results. Each line represents an instance that has been solved to optimality. Columns have the same meaning as those already described in Table 5.3. The number of communications in this case in not equal to $2 \times (n-1)$ as for the pipelined instances, but depends on the specific task graph. We can see that typically the behaviors are similar to those found when solving the pipelined instances, but we can note some instances where the number of iterations or the search time is notably higher. For example, in the last but two line the number of iterations is very high: this is due to the particular structure of the task graph; in fact it can happen that a high degree of parallelism between the tasks, that is a high number of tasks that can execute only after a single task, leads to a number allocations that are not schedulable. The master problem solver thus looses time proposing to the scheduler a high number of infeasible allocations. On the contrary, in the last line the number of iterations is low but the search time is extremely high: this is due to the tasks characteristics that make the scheduling problem very hard to be solved.

In addition, we have tested our approach on a DSVP instance considering decreasing values of the deadline, so as to have instances with different constraint tightness. We noticed that the phase transition of the problem happens when the deadline is not too tight to have few solutions (among which is easy to find the optimal one) and not too loose so as the problem is trivially solvable assigning all tasks to the same processor at the lowest speed. In addition, varying the deadline constraints, the best and the worst search time remain within an order of magnitude, so our methodology efficiently faces instances with different densities of feasible solutions.

Finally, we intended to compare computation efficiency of our hybrid approach with that of traditional approaches not leveraging problem decomposition (i.e., the whole mapping problem modelled through ILP or CP). However, such a comparison was already reported in 4 for a simpler problem (power con-

Activities				
Alloc	**Sched**	**Procs**	**Time(s)**	**Iters**
8+12	8+12	3	1,48	2
8+12	8+12	3	4,26	6
8+16	8+16	2	1,57	1
8+16	8+16	3	0,81	1
8+16	8+16	4	0,86	1
9+8	9+8	2	2,73	3
9+10	9+10	4	2,60	4
9+12	9+12	4	1,40	3
9+12	9+12	4	2,14	5
9+12	9+12	2	1,11	1
9+16	9+16	3	35,95	43
9+16	9+16	4	29,59	26
9+16	9+16	4	4,84	6
9+20	9+20	3	2,51	1
9+20	9+20	6	158,43	39
9+22	9+22	3	6,62	2
9+24	9+24	2	2,51	1
10+12	10+12	4	0,37	1
10+12	10+12	4	11,50	27
10+16	10+16	3	12,81	3
10+16	10+16	4	13,92	14
10+18	10+18	2	5,90	1
10+18	10+18	3	2,12	1
10+24	10+24	4	4,18	5
12+20	12+20	5	551,92	213
14+22	14+22	2	14,11	1
14+62	14+62	6	3624,81	2

Table 5.4: Search time and number of iterations for instances with generic task graphs

sumption was not accounted for, only performance was), and already showed a computation efficiency gap of orders of magnitude. Considering an upper bound of 15 minutes for the search time, CP and IP proved capable of finding the optimal solution only for extremely small instances, with a low number of tasks and PEs, and to find a solution (not the optimal one) only in 50% of the hard instances, while the hybrid approach solved 100% of the instances to optimality.

5.10 Design time support

A software development and optimization flow based on the above hybrid solver addresses the optimality gap usually incurred by fast exploration frameworks. On the other hand, this flow requires a correspondent design-time and run-time support in the target platform matching the way the application and the architecture have been abstracted in the optimization framework and allowing the precise implementation of computed mapping solutions. In practice, such support is needed to close the abstraction gap (i.e., the deviation between the mapping problem model and the real behavior of the target platform), which is the other main objective of this chapter.

5.10.1 Application and task computational model

Our methodology requires to model the multi-task application to be mapped and executed on top of the target HW platform as a task graph with precedence constraints. Nodes of the graph represent concurrent tasks while the arcs indicate mutual dependencies (communication and/or synchronization).

INPUT
EXEC
OUTPUT

Figure 5.6: Three phases behavior of Tasks.

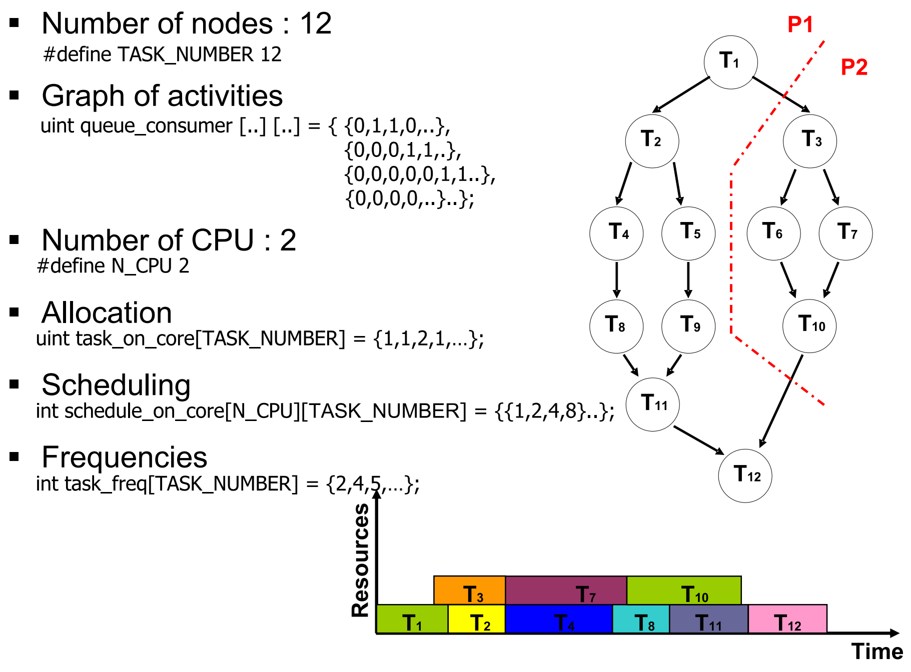
Task execution is structured in three phases, as indicated in Fig.5.6: all input communication queues are read (INPUT), computation activity is performed (EXEC) and finally all output queues are written (OUTPUT). Each phase consists of an atomic activity. Each task also has 2 kinds of associated memory requirements:

- Program Data: storage locations are required for computation data and for processor instructions;
- Communication queues: the task needs queues to transmit and receive messages to/from other tasks, eventually mapped on different processors.

Program data are allocated on the private memory of each processor, while communication queues reside in scratch-pad memory (in case the communicating tasks run on the same processor) or in shared memory (for remote communications).

5.10.2 Customizable Application Template

We set up a generic customizable application template allowing software developers to easily and quickly build their parallel applications starting from a high-level task and data flow graph specification. Programmers can at first think about their applications in terms of task dependencies and quickly draw the task graphs, and then use our tools and libraries to translate the abstract representation into C-code. This way, they can devote most of their effort to the functionality of tasks rather than the implementation of their communication, sychronization, scheduling and energy mode switching mechanisms. Following our scalable and parameterizable template, we also ensure that the final implementation of the target application will be compliant with the modelling assumptions of the optimizer, and that the optimal performance and the constraint satisfaction of computed mapping solutions will be achieved in practice.



- Number of nodes : 12

```
#define TASK_NUMBER 12
```

- Graph of activities

```
uint queue_consumer [..] [..] = { {0,1,1,0,..},
                                  {0,0,0,1,1,.},
                                  {0,0,0,0,0,1,1..},
                                  {0,0,0,0,..}..};
```

- Number of CPU : 2

```
#define N_CPU 2
```

- Allocation

```
uint task_on_core[TASK_NUMBER] = {1,1,2,1,...};
```

- Scheduling

```
int schedule_on_core[N_CPU][TASK_NUMBER] = {{1,2,4,8}..};
```

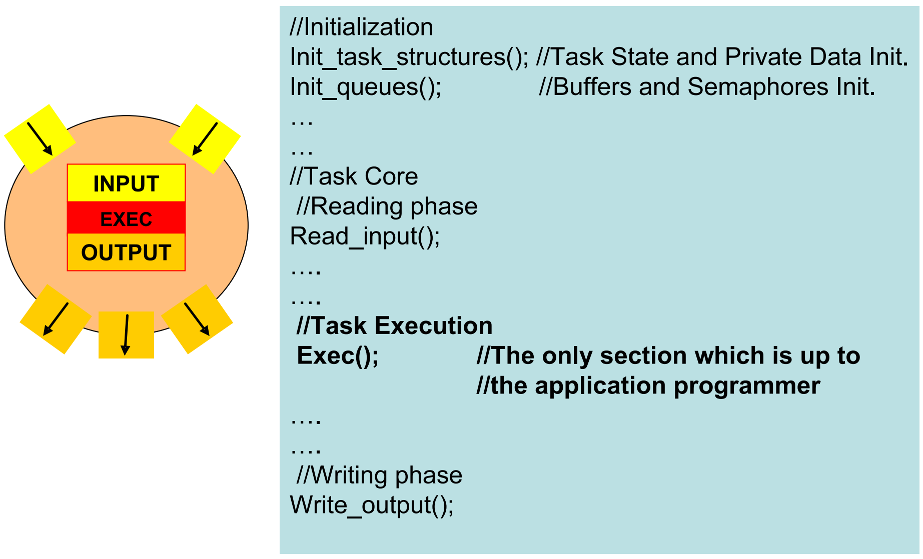
- Frequencies

```
int task_freq[TASK_NUMBER] = {2,4,5,...};
```

Figure 5.7: Example of how to use the Customizable Application Template.

Fig.5.7 shows a pictorial illustration of how our template looks like. Programmers can specify the structure of the target application by simply declaring a series of macros and data structures. In the example, we have depicted a task graph with twelve tasks and with precedence constraints defined in the matrix *queue_consumer[][]*. If task i has a precedence constraint w.r.t. task j, the element queue_consumer[i][j] will be set to 1. Developers can also specify information about the configuration of the target hardware platform and the desidered allocation and schedule, as derived from the optimization tool. *N_CPU macro* specifies the number of available processing cores. The two *task_on_core[]* and *schedule_on_core[][]* data structures specify where tasks should be allocated and which schedule to apply for each core, while with the *task_freq[]* vector developers can associate an operating voltage/frequency pair to each task.

For every task indicated within the application template, C-code is auto-



```
//Initialization
Init_task_structures(); //Task State and Private Data Init.
Init_queues();            //Buffers and Semaphores Init.
…
…
//Task Core
 //Reading phase
Read_input();
….
….
 //Task Execution
 Exec();             //The only section which is up to
                     //the application programmer
….
….
 //Writing phase
Write_output();
```

Figure 5.8: Task computational versus the C-code generated.

matically generated. Fig.5.8 shows C-code for a task reflecting the considered computation model. At task creation, the task state and private data structures are instantiated and initializated, as well as all buffers and semaphores needed for communication and synchronization. The INPUT phase of the computational model corresponds to the *Read_input()* function, while the OUTPUT phase to the *Write_output()* one. These two functions are blocking and handle the whole communication and synchronization procedures automatically. The only section which is on burden of the programmer is the *Exec()* function: this is the customizable computational core of the task.

5.11 Run-time support

We implemented a set of APIs by which users can easily reproduce optimizer solutions on their target platform with great accuracy.

5.11.1 OS-independent allocation and scheduling support

Once the target application has been implemented using our generic customizable template, tasks, program data and communication queues are allocated to the proper hardware resources (processor or memory cores) as indicated by the computed allocation solution. This is done through the *init_task* of our template which allocates and launches all the activities at booting time.

In order to reproduce the exact scheduling behavior of the optimizer, we implemented a scheduling support middleware in the target platform. Using this facility, programmers only have to specify the desired scheduling for every processor core, which is handled accordingly by our middleware in a transparent way.

After the boot of the application, our framework sets to active only the first task in the scheduling list, while the other ones are set to the sleep state. In this way, we avoid any undesired task preemption by the OS scheduler, which would induce a different behavior with respect to the optimal one provided by the optimizer.

After the active task has finished its execution, it is put to sleep thus releasing the CPU, while the subsequent task in the scheduling list is woken up by switching its state to active. If the subsequent task is allocated to a different CPU, this remote wake up mechanism is handled via interrupts. Every time a new task is scheduled, our middleware sets its right operating frequency as specified in the application template.

5.11.2 Communication and Synchronization support

Software support for efficient messaging is also provided by our set of high-level APIs. The communication and synchronization library abstracts low level architectural details to the programmer, such as memory maps or explicit management of hardware semaphores or interrupt signaling. The structure of the queues is shown in Fig. 5.9.

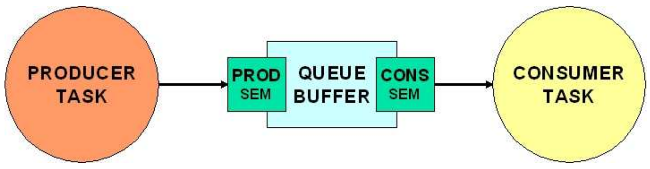


Figure 5.9: The structure of a queue.

The infrastructure for the communication between a producer-consumer pair is composed by a data queue and by two semaphores. The communication and synchronization mechanisms have already been illustrated in Section 6.4. If one task has got more than one input or output queue, our optimizer can specify the optimal reading/writing sequence from/to them. We tuned our run-time support to enable this option. This is a very important feature, since an optimal queue-usage ordering can increase the parallelism and thus boost performance. Fig. 5.10 better clarifies this issue. It shows a case study in which six tasks are allocated to two different processing cores.

Task T1 has to communicate with both T2 and T3, which are allocated to the same core, and with T4 allocated to a different core. At start-up, let us assume that task T1 will be scheduled on CPU1 and task T4 on CPU2. While T1

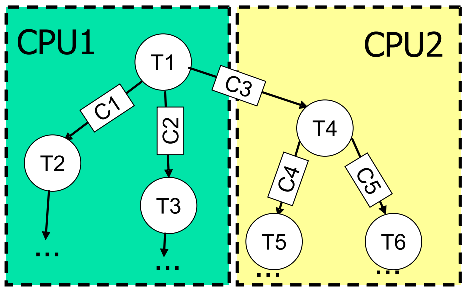


Figure 5.10: Optimal queue usage ordering: example.

immediately starts its execution, T4 has to wait for data from T1, thus keeping CPU2 stalled. The idle wait of T4 depends on the queue-fill ordering enforced by T1: it will be shorter if T1 gives maximum priority to queue C3. Both our optimization framework and our application execution support can handle this additional degree of freedom for performance optimization.

5.12 Experimental Results

For each task in the input graph the optimizer needs the worst-case execution time, the time required for writing and for reading input data from local memory and the overhead for writing and reading input data if queues are allocated onto the shared memory in the absence of contention. For each task graph, this information can be collected with only 2 simulation runs on a virtual platform. As mentioned in Section 6.4, we used the MPARM platform for complete MPSoC functional simulation with clock-cycle accuracy [26] in SystemC. This modelling and simulation environment was used both to provide input data for the optimization framework and to validate its solutions.

Two types of *validation experiments* were performed, namely (i) comparison of simulation-based energy and throughput with optimizer-derived values, and (ii) prove of viability of the proposed approach for real-life demonstrators (GSM, JPEG).

5.12.1 Validation of optimizer solutions

We have deployed the virtual platform to implement the allocations, schedules and frequency assignments generated by the optimizer. A tunable multitask application has been used for this experiment, allowing to change system and application parameters (local memory size, execution times, data size, real-time requirements, etc.) and to generate the 200 problem instances used for validation. The results of the validation phase are reported in Fig.5.11, which shows the distribution of energy deviations. The average difference between

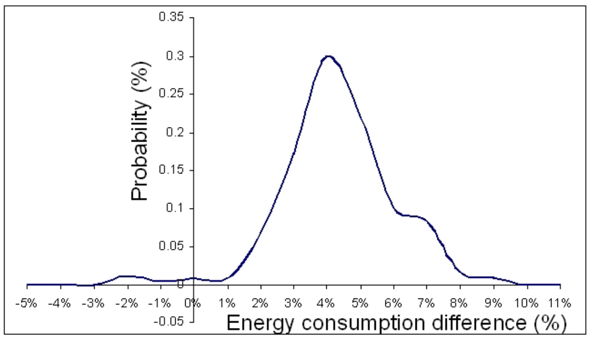


Figure 5.11: Distribution of Energy Consumption differences.

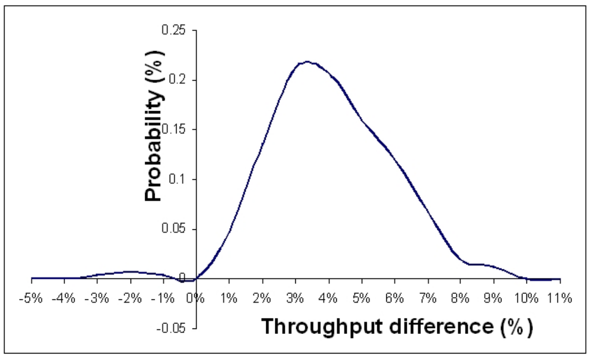


Figure 5.12: Distribution of throughput deviations.

measured and predicted energy values is 4.80%, with 1.71 standard deviation. Fig.5.12 shows the distribution of throughput differences: in this case the average difference between measured and predicted values is 4.51%, with 1.94 standard deviation. This confirms the high level of accuracy achieved by the developed optimization framework in modelling real-life MPSoCs with the assumed architectural template.

5.12.2 Demonstrators

GSM demonstrator

The methodology has been applied to a GSM codec parallelized in two ways: the first variant features 10 generic tasks while the second one consists of 6 tasks ordered in a logic pipeline. Each task has been pre-characterized by the virtual platform to provide parameters of task models to the optimizer. After the optimization stage, the solution has been validated on the virtual platform.

Fig.6.16 shows the task graph of the first GSM implementation variant. The time taken by the optimizer to come to a solution was 0.2 seconds and Table 5.12.2 shows the results of this optimization run. The validation process of the solution on the virtual platform running two cores showed an accuracy by

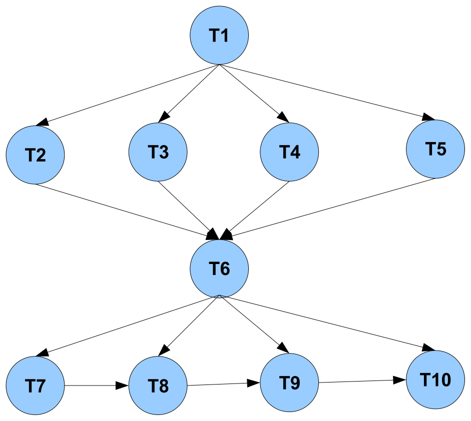


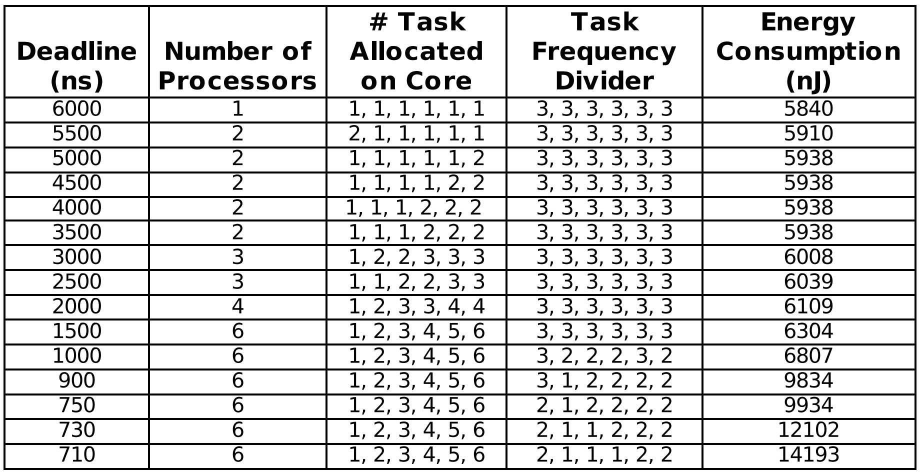
Figure 5.13: Task graph of the first GSM implementation variant with 10 tasks.

3.99% on throughput and by 2.91% on energy requirements.

Deadline (ns)	# of Proc.	Allocation of Tasks to Core	Task Freq. Divider	Energy Consump. (nJ)
5000	2	1,1,1,1,2,1,1,2,2,2	1,2,2,2,4,1,1,1,1,1	4784

Table 5.5: Mapping solution for the GSM encoder.

The results for the pipelined version of the GSM codec showed an accuracy on the processor energy dissipation, as predicted by the optimizer, by 2%. We used the pipelined version of the GSM demonstrator to explore how the optimizer minimizes energy dissipation of the processor cores with varying real-time requirements, and the results are reported in Fig.5.14. The behaviour of the optimizer is not specific for the GSM case study, but can be extended to all applications featuring timing constraints.



Deadline (ns)	Number of Processors	# Task Allocated on Core	Task Frequency Divider	Energy Consumption (nJ)
6000	1	1, 1, 1, 1, 1, 1	3, 3, 3, 3, 3, 3	5840
5500	2	2, 1, 1, 1, 1, 1	3, 3, 3, 3, 3, 3	5910
5000	2	1, 1, 1, 1, 1, 2	3, 3, 3, 3, 3, 3	5938
4500	2	1, 1, 1, 1, 2, 2	3, 3, 3, 3, 3, 3	5938
4000	2	1, 1, 1, 2, 2, 2	3, 3, 3, 3, 3, 3	5938
3500	2	1, 1, 1, 2, 2, 2	3, 3, 3, 3, 3, 3	5938
3000	3	1, 2, 2, 3, 3, 3	3, 3, 3, 3, 3, 3	6008
2500	3	1, 1, 2, 2, 3, 3	3, 3, 3, 3, 3, 3	6039
2000	4	1, 2, 3, 3, 4, 4	3, 3, 3, 3, 3, 3	6109
1500	6	1, 2, 3, 4, 5, 6	3, 3, 3, 3, 3, 3	6304
1000	6	1, 2, 3, 4, 5, 6	3, 2, 2, 2, 3, 2	6807
900	6	1, 2, 3, 4, 5, 6	3, 1, 2, 2, 2, 2	9834
750	6	1, 2, 3, 4, 5, 6	2, 1, 2, 2, 2, 2	9934
730	6	1, 2, 3, 4, 5, 6	2, 1, 1, 2, 2, 2	12102
710	6	1, 2, 3, 4, 5, 6	2, 1, 1, 1, 2, 2	14193

Figure 5.14: Behaviour of the optimizer with varying real-time requirements. Allocation is given as an array indicating the processor ID on which each task is mapped. Similarly, the frequency of each task is expressed in terms of the integer divider of the baseline frequency. Only 3 dividers are used for this example.

When the deadline is loose, all tasks are allocated to one single processor at the minimum frequency (66 MHz, corresponding to a divisor of 3). As the deadline gets tighter, the optimizer prefers to employ a second processor and to progressively balance the load, instead of increasing task frequencies. This procedure is repeated every time a new processor has to be allocated to meet the timing constraints. Only under very tight deadlines, the optimizer leverages increased task frequencies to speed-up the system. To the limit, the system works with 1 task on each processor, although not all tasks run at the maximum frequency. In fact, the GSM pipeline turns out to be unbalanced, therefore it would be energy inefficient to run the shorter tasks at maximum speed, and would not even provide performance benefits. As a result, the optimizer determines the most energy-efficient configuration that provides the best performance. The problem becomes infeasible if more stringent deadlines than 710 ns are required. We will show in the next sub-section that this optimizer behaviour is a function of the computation-communication ratio.

JPEG demonstrator

Our methodology was then applied to a JPEG decoder, which was partitioned in 4 pipelined tasks: Huffman DC decoding, Huffman AC decoding, inverse quantization, inverse DCT. Each stage processes an 8x8 block, amounting to an exchange of 1024 bit among pipeline stages. The accuracy of the energy estimation given by the optimizer was found to be 3.1% from functional simulation. In contrast to pipelined GSM, user requirements on a JPEG decoding usually consist of the minimization of the execution time and not of a deadline to be met. However, a performance-energy conflict arises, and two approaches to allocation and scheduling of a JPEG decoder task graph are feasible. On one hand, the designer could be primarily interested in reducing execution time at the cost of increased energy. On the other hand, the primary objective function could be the minimization of energy dissipation, whatever the decoding performance. This trade-off has been investigated with the optimizer and the Pareto-optimal frontier in the performance-energy space is illustrated in Fig.5.15. The constraint on the execution time on the x-axys has been translated into a constraint on the block decoding time. The curve is not linear since there is a discrete number of voltage-frequency pairs, which makes the problem for the optimizer much more complex.

As we can observe, for a large range of deadlines, the optimizer is good at improving system performance without significantly changing processor energy dissipation. This is done by using one or two processors, changing the allocations and using high frequency dividers. Beyond 200 ns, the optimizer is forced to use low frequency dividers, thus causing the energy to skyrocket. Interestingly, the increase of task frequency is preferred to an increase of the number of processors, since the communication energy would involve even higher total energy consumption. This behaviour is different from the one seen for the GSM, since this time the computation-communication ratio is lower than for GSM due a larger size of exchanged messages.

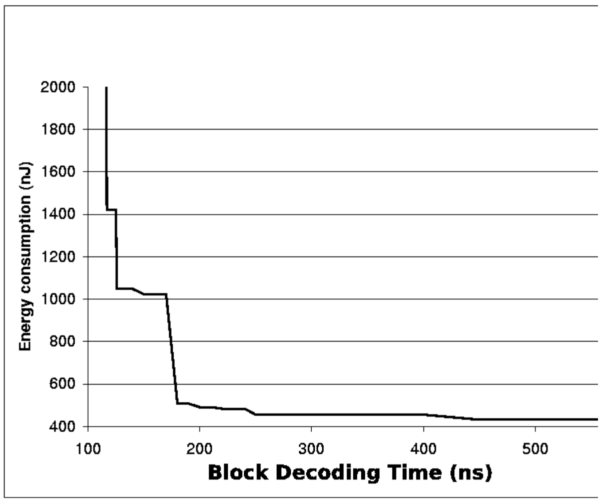


Figure 5.15: Pareto-optimal frontier in the performance-energy design space.

5.13 Comparison between optimal and heuristic approaches

In this section we illustrate a comparison of mapping and frequency/voltage assignment solutions generated by our complete solver with those provided by a heuristic approach leveraging genetic algorithms.

5.13.1 Genetic-Based Energy Optimization Heuristic

The heuristic was taken from [33]. Originally, the approach presented in [33] associates a communication task that has to be scheduled for each message exchanged over the bus. In order to have a fair comparison with our approach, we have implemented the additive bus model used in this chapter.

The optimization flow of the heuristic is split in three parts:

- Genetic task allocation optimization
- Genetic schedule optimization
- Optimal frequency selection

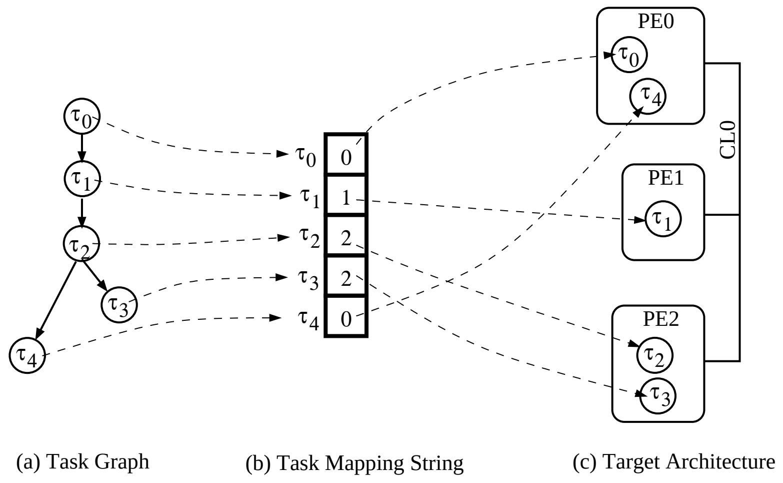


Figure 5.16: Task allocation string describing the mapping of five tasks to an architecture

In the genetic task allocation approach, solution candidates are encoded into allocation strings, as shown in Fig. 5.16. Each gene in these strings describes a candidate allocation of a task to a processor. For instance, task τ_4 in Fig. 5.16 is mapped to the processing element PE0. As typical in all genetic algorithms, ranking, selection, crossover, mutation and offspring insertion are applied in order to evolve an initial solution pool. The key feature of this algorithm is the invocation of a genetic scheduling procedure for each allocation candidate, in order to calculate the fitness function that guides the optimization.

The genetic scheduling algorithm finds, for a given allocation, the schedule that meets all the task deadlines and, furthermore, has the minimum energy. We deployed one of the most widely used heuristic approaches to scheduling, namely list scheduling, which takes scheduling decisions based on task priorities.

Clearly, different assignments of priorities result in different schedules. The task priorities are encoded into a priority string. The genetic algorithm aims at finding an assignment of priorities that leads to a schedule of high quality in terms of timing behaviour and exploitable slack time. Both crossover and mutation are applied during the iterative execution of the genetic algorithm. The algorithm terminates after a stop criterion is fulfilled (for example, a bound of the number of consecutive generations that did not improve significantly the solution).

A fitness function is used for evaluating the quality of a schedule. The fitness function captures the energy of a certain schedule. After the list scheduling has constructed a schedule for a given set of priorities, the algorithm proceeds by passing this schedule to a voltage selection algorithm that identifies the task voltages that minimize the energy dissipation. After performing the voltage selection, the fitness F_S of each schedule candidate is calculated.

As we have seen, the voltage selection is the core of the global energy optimization. During the genetic scheduling step, a fast voltage scaling heuristic is used. However, once the genetic algorithms are completed, we use the optimal frequency scaling algorithm presented in [34], restricted to select one single frequency for each task. Consequently, if the genetic heuristic method finds the best allocation and schedule, after the last frequency selection step, we will obtain a globally optimal result, identical to the one produced by the approach proposed in our work.

5.13.2 Comparison results

The complete and the heuristic algorithms have been put at work with a number of task graphs featuring from 10 to 20 tasks each. We found that the heuristic approach never provides the optimal solution for the problem instances under test. Nevertheless, in 70% of the cases energy consumption difference is within 5%, but in the 10% of the cases it is extremely high (up to 44%). The energy consumption of generated mappings differs on average by 8.02%, with a standard deviation of 15.94. The minimum difference is 0.01%, while the maximum is 43.76%.

By setting loose search stopping criteria for the heuristic method (thus giving it more time to optimize the solution), we allowed this latter to take a search time comparable to that of our technique. In spite of this, our complete method is able to find optimal solutions that the heuristic algorithm is not able to find. Since our approach extends the applicability of complete methods to large problem instances at an affordable search time for statically scheduled systems, this experiment further confirms the distinctive advantage of doing this, namely the generation of power-efficient system configurations which heuristic methods hardly provide.

Although we do not have any performance-related contribution to the objective functions in both methods, we found it interesting to compare the execution time of applications with the mapping solutions under test, viewing it as a side effect of the optimization process. Obviously, real-time constraints are always met in all solutions. We got a mean difference between execution time values of 10.59%, standard deviation 12.02%, minimum -1.94% and maximum 29.11%. When the execution time difference is negative, it means that the makespan found by the heuristic algorithm is lower with respect to that of the hybrid approach. This happens in 29% of the cases. This means that when the heuristic method provides lower execution times, it does that at the cost of employing lower frequency dividers, thus incurring higher energy. In contrast, sub-optimal mappings and/or schedulings may lead to longer execution times and and to higher energy values as well.

In the second experiment we performed, we solved a common problem instance while varying the real time constraint, i.e., we varied the deadline value (a common value for processor and task deadlines was chosen) from a very loose one (allowing all tasks to run at the lowest speed on the same processor) to the tightest one.

Results are shown in Figure 5.17: we report the energy consumption (y axis) of the solutions found by the complete and the heuristic approach as a function of the deadline (x axys). We can see that the heuristic solution is never the optimum, even when the real-time constraint is weak. In fact, the relative difference for the loosest deadline value is equal to 0.55%. Such difference then grows as the real-time constraint becomes tighter, and for deadline values lower than 3ms the heuristic approach is not even able to find a solution, while the complete solver finds that the lowest possible deadline value is around 2.1ms. We find this capability of our solver to extend the range of problem feasibility very interesting from an application viewpoint.

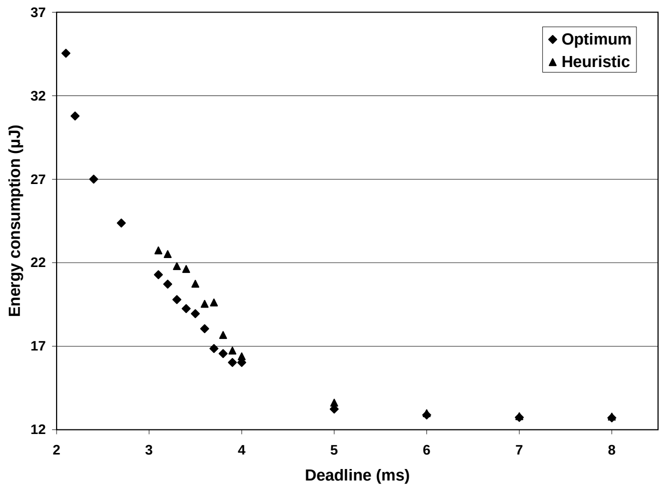


Figure 5.17: Comparison between optimal and heuristic solutions

5.14 Conclusions

In this chapter, we address both the optimality gap and the abstraction gap which impair the results of traditional software optimization flows for on-chip multiprocessor systems. On one hand, we present a cooperative framework to solve the allocation, scheduling and voltage/frequency selection problem to optimality for energy-efficient MPSoCs. This iterative framework is based on Logic-Based Benders Decomposition and provides optimal solutions at an affordable search time, orders of magnitude shorter than traditional ILP or CP solvers. On the other hand, we set up a design-time and a run-time support for the target MPSoC platform allowing to specify applications while matching optimizer modelling assumptions and to exactly implement mapping solutions computed by the hybrid solver, thus achieving expected performance results and constraint satisfaction. Interestingly, we automate most of the steps for software development and optimization, thus allowing programmers to concentrate on functionality and code optimization and not on the application execution support. Experimental results confirm the high accuracy of optimizer solutions as validated with cycle-accurate functional simulation. Finally, a comparison of our approach with a heuristic algorithm proves the capability of our solving framework to find solutions even in limit operating conditions and to save a significant amount of energy in many problem instances due to the computation of the optimal solution with satisfactory search times.

Bibliography

[1] Horowitz, M., Alon, E., Patil, D., Kumar, S.N.R., Bernstein, K.: Scaling, power, and the future of cmos. In: IEEE Intl. Electron Devices Meeting (IEDM 2005). (2005) 9–15

[2] Ho, R., K.Mai, Horowitz, M.: The Future of Wires. Proc. IEEE **89** (2001) 490–504

[3] Brodersen, R.W., Horowitz, M.A., Markovic, D., Nikolic, B., Stojanovic, V.: Methods for true power minimization. In: Intl. Conf. on Computer-Aided Design (ICCAD 2002). (2002) 35–42

[4] Mudge, T.: Power: A First-class Architectural Design Constraint. IEEE Computer **34** (2001) 52–58

[5] S.Borkar: Thousand core chips: a technology perspective. In: Design Automation Conference. (2007) 746–749

[6] Krewell, K.: Best servers of 2004. In: Microprocessor Report. (2005)

[7] G.Martin: Overview of the mpsoc design challenge. In: Design Automation Conference. (2006) 274–279

[8] Benini, L., Bertozzi, D., Guerri, A., Milano, M.: Allocation and scheduling for mpsocs via decomposition and no-good generation. In: Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP2005). Volume 3709 of Lecture Notes in Computer Science., Springer (2005) 107–121

[9] S.Medardoni, M.Ruggiero, D.Bertozzi, L.Benini, G.Strano, C.Pistritto: Capturing the interaction of the communication, memory and i/o subsystems in memory-centric industrial mpsoc platforms. In: Design Automation and Test in Europe Conf. (2007) 1–6

[10] H.Kazuyuki, J.: ARM MPCore: The Streamlined and Scalable ARM11 Processor Core. ASP-DAC07 (2007) 748–748

[11] et al., N.: Deterministic Inter-Core Synchronization with Potentially All-in-Phase Clocking for Low-Power Multi-Core SoCs. In: ”Int. Solid-State Circuits Conference, Digest of Technical Papers”. (2005) 296–299

[12] : Freescale Technologies for Energy Efficiency: 2007 Overview. In: ”White Paper”. (2007)

[13] Andrei, A., Schmitz, M., Eles, P., Peng, Z., Al-Hashimi, B.: Overhead-Conscious Voltage Selection for Dynamic and Leakage Power Reduction of Time-Constraint Systems. In: ”Proc. Design, Automation and Test in Europe (DATE)”. (2004) 518–523

[14] Kwok, Y., Ahmad, I.: Static Scheduling Algorithms for allocating directed task graphs to multiprocessors. ACM Computing Surveys **31** (1999) 406–471

[15] Eles, P., Peng, Z., Kuchcinski, K., Doboli, A., Pop, P.: Scheduling of conditional process graphs for the synthesis of embedded systems. In: Design, automation and Test in Europe. (1998) 132–139

[16] Ventroux, N., Blanc, F., Lavenier, D.: A low complex scheduling algorithm for multi-processor system-on-chip. In: Parallel and Distributed Computing and Networks. (2005) 540–545

[17] Axelsson, J.: Architecture synthesis and partitioning of real-time synthesis: a comparison of 3 heuristic search strategies. In: Procs. of the 5th Intern. Workshop on Hardware/Software Codesign (CODES/CASHE97). (1997) 161–166

[18] Hooker, J.N., Ottosson, G.: Logic-based benders decomposition. In: Mathematical Programming. (2003) 33–60

[19] by M.Milano, E.: Constraint and Integer Programming: Toward a unified methodology (Operations Research/Computer Sciences Interfaces). Kluwer Academic Publisher (2003)

[20] Yao, F., Demers, A., Shenker, S.: A Scheduling Model for Reduced CPU Energy. In: IEEE Symposium on Foundations of Comp. Science. (1995) 374–382

[21] Ishihara, T., Yasuura, H.: Voltage Scheduling Problem for Dynamically Variable Voltage Processors. In: ”Proc. Int. Symp. Low Power Electronics and Design (ISLPED)”. (1998) 197–202

[22] Xie, F., Martonosi, M., Malik, S.: Bounds on power savings using runtime dynamic voltage scaling: an exact algorithmand a linear-time heuristic

approximation. In: ”Proc. Int. Symp. Low Power Electronics and Design (ISLPED)”. (2005) 287–292

[23] R. Jejurikar, R.G.: Dynamic Slack Reclamation with Procrastination Scheduling in Real-Time Embedded Systems. In: Design Automation Conference (DAC). (2005)

[24] Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the theory of NP-Completeness. W.H. Freeman and Company (1979)

[25] Schmitz, M.T., Al-Hashimi, B.M., Eles, P.: Iterative Schedule Optimization for Voltage Scalable Distributed Embedded Systems. ACM Transactions on Embedded Computing Systems **3** (2004) 182–217

[26] Hu, J., Marculescu, R.: Energy-aware communication and task scheduling for network-on-chip architectures under real-time constraints. In: ”Proc. Design, Automation and Test in Europe (DATE)”. (2004) 234–239

[27] Benini, L., Bertozzi, D., Guerri, A., Milano, M.: Allocation and Scheduling for MPSoCs via Decomposition and No-Good Generation”, booktitle = ”Proc. of International Joint Conference on Artificial Intelligence (IJCAI). (2005) 1517–1518

[28] Gruian, F., Kuchcinski, K.: LEneS: Task Scheduling for Low-Energy Systems Using Variable Supply Voltage Processors. In: ”Proc. ASP-DAC’01”. (2001) 449–455

[29] Martin, S., Flautner, K., Mudge, T., Blaauw, D.: Combined Dynamic Voltage Scaling and Adaptive Body Biasing for Lower Power Microprocessors under Dynamic Workloads. In: ”Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)”. (2002) 721–725

[30] Leung, L.F., Tsui, C.Y., Ki, W.H.: Minimizing Energy Consumption of Multiple-Processors-Core Systems with Simultaneous Task Allocation, Scheduling and Voltage Assignment. In: ”Proc. of Asia South Pacific Design Automation (ASP-DAC)”. (2004) 647–652

[31] M.Ruggiero, A.Acquaviva, D.Bertozzi, L.Benini: Application-specific power-aware workload allocation for voltage scalable mpsoc platforms. In: Proc. of ICCD 2005. (2005) 87–93

[32] F.Poletti, A.Poggiali: Flexible hardware/software support for message passing on a distributed shared memory architecture. In: Proc. of DATE Conf. (2005) 736–741

[33] Schmitz, M.T., Eles, P., Al-Hashimi, B.M.: System-Level Design Techniques for Energy-Efficient Embedded Systems. Kluwer Academic Publisher (2004).

[34] Andrei, A., Eles, P., Peng, Z., Schmitz, M., Al-Hashimi, B.M.: Energy Optimization of Multiprocessor Systems on Chip by Voltage Selection. In: IEEE Transactions on Very Large Scale Integration Systems, vol.15, no.3, pp. 262275, 2007.

Chapter 6

Communication-Aware Stochastic Allocation and Scheduling Framework for Conditional Task Graphs in Multi-Processor Systems-on-Chip

6.1 Overview

Designers, thanks to the increasing levels of system integration, are turning to multicore architectures to satisfy the ever-growing computational needs of applications within a reasonable power envelope. One of the most interesting challenges for MultiProcessor System-on-Chip (MPSoC) success is the development of new design flows for efficient mapping of multi-task applications onto hardware platforms. Even though data-flow graphs are often used for pure data-streaming, many realistic applications can only be specified as conditional task graphs (CTG). The problem of allocating and scheduling conditional task graphs on processors in a distributed real-time system is NP-hard.

The first contribution of this chapter is a complete stochastic allocation and scheduling framework, where an MPSoC virtual platform is used to accurately derive input parameters, validate abstract models of system components and assess constraint satisfaction and objective function optimization. The optimizer implements an efficient and exact approach to allocation and scheduling based on problem decomposition. The original contributions of the approach appear both in the allocation and in the scheduling part of the optimizer. For the first, we propose an exact analytic formulation of the stochastic objective function based on the task graph analysis, while for the scheduling part we extend the timetable constraint for conditional activities.

The second contribution of this chapter is the introduction of a software library and API for the deployment of conditional task graph applications onto Multi-Processor System-on-Chips. With our library support, programmers can quickly develop multi-task applications which will run on a multi-core architecture and can easily apply the optimal solution found by our optimizer. The proposed programming support manages OS-level issues, such as task allocation and scheduling, as well as task-level issues, like inter-task communication and synchronization.

6.2 Introduction

During last five years we lived through a paradigm shift in the design of integrated architectures from conventional single processor systems to multiprocessors. This shift has been caused by the evidence that popular approaches to maximize single processor performance have reached their limits. Prohibitive power consumption, the higher bound of boosting clock frequencies of monolithic processor cores [1] [2] [3] [4] and design complexity are the most critical factors that limit performance scaling [5].

However, Moore's law continues to enable a doubling in the number of transistors on a single die every 18-24 months. Consequently, designers are turning to multicore architectures to satisfy the ever-growing computational needs of applications within a reasonable power envelope [6]. Keeping with Moore's law, the semiconductor roadmap foresees a doubling in the number of core units per die with every process generation [7]. This trend is noticeable both in mainstream [49] [50] [39] [52], as well as in embedded computing [42] [45] [51] [53].

Future Multi-Processor System-on-Chips (MPSoCs) hosting a huge number of processors will guarantee high computational power thanks to their massive parallelism, but at the cost of a more complicated parallel programming paradigm.

One of the most daunting challenges to the success of MPSoC platforms consists of developing effective software optimization tools that can optimally exploit the available cores [8]. If we consider that software running on multiprocessor must be high performance, real-time, and low power incoming

MPsoC platforms will lead to several and interesting challenges in software development. This statement becomes more strong by the fact that consumer applications are characterized by tight time-to-market constraints and extreme cost sensitivity. We can formilize that there is a clear need for new deployment technologies which address multi processing issues in embedded systems.

Modern applications, particularly in the embedded domain, exhibit a lot of concurrency at different levels of granularity. Task level concurrency is the coarsest grained category and is exhibited when the computation contains multiple flows of control. The key to successful application deployment lies in effectively mapping this concurrency in the application to the architectural resources provided by the platform. Even though data-flow graphs are often used for pure data-streaming applications, many realistic applications can only be specified as conditional task graphs. Mapping a multi-task application to a multi-core architecture is a key step of the software development flow, as it significantly impacts design quality metrics like execution time, throughput and power. The problem of allocating and scheduling conditional task graphs on processors in a distributed real-time system is NP-hard. In addition, the intricacies of component interactions in multicore architectures call for detailed system models and for their validation on a real or virtual platform [9].

Model simplification is often achieved by abstracting away platform implementation details. As a result, optimization problems become more tractable, even reaching polynomial time complexity [10]. Unfortunately, this approach creates an abstraction gap between the optimization model and the real HW-SW platform. Neglecting this gap can generate unpredictable behaviours, like undesired system-level interactions of many concurrent execution flows. In the application developing phase, programmers must be conscious about simplified assumptions taken into account in optimization tools. For instance, a communication or synchronization sub-optimal task implementation leads to reduced throughput and/or latency and has also energy implications, due to the higher occupancy condition for system resources. Validation is therefore required and the accuracy of the solutions must be carefully assessed through detailed simulation runs or execution on the target hardware.

Moving from these considerations, in this chapter we present a novel framework for developing, allocating and scheduling conditional multi-task graphs on multi-processor systems-on-chip. We target a general template for distributed memory embedded systems where the communication architecture is becoming a critical component. Interaction of multiple traffic patterns on the system bus causes congestion and hence unpredictable communication latencies. Neglecting this behaviour in high level optimization tools for allocation and scheduling might lead to unacceptable deviations of real performance metrics with respect to predicted ones and to the violation of real-time constraints.

Our allocation and scheduling framework is based on problem decomposition and deploys techniques mutuated from the Artificial Intelligence and the Operations Research community: the allocation subproblem is solved through Integer Programming while the scheduling one through Constraint Programming. More interestingly, the two solvers can interact with each other by means of no-good generation, thus building an iterative procedure which has been proven to converge producing the optimal solution.

We propose two main contributions in this field: the first concerns both the allocation and scheduling components. The objective function we consider in the allocation component depends on the allocation variables. Clearly, having conditional tasks, the exact value of the communication cost cannot be computed. Therefore our objective function is the expected value of the communication cost. We propose here to identify an analytic approximation of this value. The approximation is based on the Conditional Task Graph analysis for identifying two data structures: the activation set of a node and the coexistence set of two nodes. The approximation turns out to be exact and polynomial. Concerning the scheduling, we propose an extension of the time-table constraint for cumulative resources, taking into account conditional activities. The propagation is polynomial if the task graph satisfies a condition called *Control Flow Uniqueness* which is quite common in many conditional task graphs for system design. To address the abstraction gap, we formulated an accurate model for allocation and scheduling, which accounts for a number of non-idealities in real-life hardware platforms.

The other main contribution of this chapter is the introduction of a new methodology for multi-task application development. We present the implementation of the static (design-time) and dynamic (run-time) software infrastructure required to deploy the applications on the target platform. This is a critical and non-trivial task, as we must guarantee that actual execution accurately matches in time and space the solution computed by the optimizer. We propose a software library and APIs for the deployment of conditional task graph applications onto Multi-Processor System-on-Chips. The proposed programming support manages OS-level issues, such as task allocation and scheduling, as well as task-level issues, like inter-task communication and synchronization. We carried out its implementation with both high flexibility and performance in mind.

Finally, we deploy an MPSoC virtual platform to validate the results of the optimization steps and to more accurately assess constraint satisfaction and objective function optimization. In multi-processor systems, we believe this validation phase is critical in order to check modelling assumptions and make sure that second-order effects and/or modelling approximations impair optimizer-predicted performance (e.g., a required throughput) only marginally below 10%.

The structure of this work is as follows. Section II illustrates previous work. Section III presents the target architecture while high level application and system models, simplifying the optimization framework, are reported in Section IV. Our combined solver for the mapping problem is described in Section V, its computation efficiency in Section VI and its integration in a software optimization methodology for MPSoCs in VII. Section VIII finally shows experimental results.

6.3 Related Work

The synthesis of system architectures has been extensively studied in the past. Mapping and scheduling problems on multi-processor systems have been traditionally tackled by means of Integer Linear Programming (ILP). In general, even though ILP is used as a convenient modelling formalism, there is consensus on the fact that pure ILP formulations are suitable only for small problem instances, i.e. task graphs with a reduced number of nodes, because of their high computational cost. An early example is represented by the SOS system, which used mixed integer linear programming technique (MILP) [40]. A MILP model that allows to determine a mapping optimizing a trade-off function between execution time, processor and communication cost is reported in [17].
The complexity of pure ILP formulations for general task graphs has led to the deployment of heuristic approaches. Heuristic approaches provide no guarantees about the quality of the final solution, and many times the need to bound search times limits their applicability to moderately small task sets. In [22] a retiminig heuristic is used to implement pipelined scheduling, while simulated annealing is used in [38]. A comparative study of well-known heuristic search techniques (genetic algorithms, simulated annealing and tabu search) is reported in [13]. Unfortunately, busses are implicit in the architecture, unlike in [24]. A scalability analysis of these algorithms for large real-time systems is introduced in [32]. Many heuristic scheduling algorithms are variants and extensions of list scheduling [23]. In general, scheduling tables list all schedules for different condition combinations in the task graph, and are therefore not suitable for control-intensive applications.
Constraint Logic Programming (CP) is an alternative approach to Integer Programming for solving combinatorial optimization problems [34]. The work in [43] is based on Constraint Logic Programming to represent system synthesis problem, and leverages a set of finite domain variables and constraints im-

posed on these variables. Both ILP and CP techniques can claim individual successes but practical experience indicates that neither approach dominates the other in terms of computational performance. The development of a hybrid CP-IP solver that captures the best features of both would appear to offer scope for improved overall performance. However, the issue of communication between different modelling paradigms arises. One method is inherited from the Operations Research and is known as Benders Decomposition [18]: it is has been proven to converge producing the optimal solution. There are a number of papers using Benders Decomposition in a CP setting [46] [25] [31] [8] [9].

[41] presents an approach leverages a decomposition of the problem in the context of MPSoC systems. The authors tackle the mapping sub-problem with IP and the scheduling one with CP. The work considers only pipelined streaming applications and does not handle conditional task graphs. In order to solve the problem of allocating and scheduling a general conditional task graph onto a MPSoC, the introductions of more complex problem models and cost functions, such as more complex subproblem relaxations and Benders cuts are needed.

In the system design community, the problem of allocating and scheduling a conditional multi-task application is extremely important and many researchers have worked extensively on it, mainly with incomplete approaches: for instance in [48] a genetic algorithm is devised on the basis of a conditional scheduling table whose (exponential number of) columns represent the combination of conditions in the CTG and whose rows are the starting times of activities that appear in the scenario. The number of columns is indeed reasonable in real applications. The same structure is used in [33], which is the only approach that uses Constraint Programming for modelling the allocation and scheduling problem. Indeed the solving algorithm used is complete only for small task graphs (up to 10 activities). Besides related literature for similar problems, the Operations Research community has extensively studied stochastic optimization in general. The main approaches are: sampling [12] consisting in approximating the expected value with its average value over a given sample; the *l-shaped* method [35] which faces two phase problems and is based on Benders Decomposition [18]. The master problem is a deterministic problem for computing the first phase decision variables. The subproblem is a stochastic problem that assigns the second phase decision variables minimizing the average value of the objective function. A different method is based on the branch and bound extended for dealing with stochastic variables, [37].

The CP community has recently faced stochastic problems: in [47] stochastic constraint programming is formally introduced and the concept of solution is replaced with the one of *policy*. In the same paper, two algorithms have been proposed based on backtrack search. This work has been extended in [44] where an algorithm based on the concept of scenarios is proposed. In particular, the paper shows how to reduce the number of scenarios, maintaining a good expressiveness.

6.4 Target Architecture

Our mapping strategy targets a general template for a message-oriented distributed memory architecture. An embodiment of this template architecture is considered in this work, in order to be able to provide input data to the optimization framework, to valide its solutions based on functional simulation and to validate objective function values. The specific platform instance, conforming to the template, only determines the annotated values in the application task graph (cost for communication and execution times), which is an input to our framework. However, alternative architectures matching the same template can be input to our methodology, with just the burden to re-characterize the costs for basic communication and synchronization mechanisms, Therefore, the allocation and scheduling methodology we propose is not affected by specific design choices (e.g., the kind of processing unit, the bus architecture).

The characteristics of the architectural template targeted by our optimization framework include:

1. support for message exchange between the computation tiles,
2. availability of local memory devices at the computation tiles and of remote (i.e., non-local to the tiles, accessible through the system bus) storage devices for those program data that cannot be stored in local memories.

The remote storage can be provided by a unified memory with partitions associated with each processor or by a separate private memory for each processor core connected to the system bus. This assumption concerning the memory hierarchy reflects the typical trade-off between low access cost, low capacity local memory devices and high cost, high capacity memory devices at a higher level of the hierarchy.

We deployed the model of an instance of this architectural template in order to prove the viability of our approach (see Fig. 6.1). The computation tiles are supposed to be homogeneous and consist of ARM cores (including instruction and data caches) and of tightly coupled software-controlled scratchpad memories for fast access to program operands and for storing input data. We used an AMBA AHB bus as system interconnect. A DMA engine is attached to each

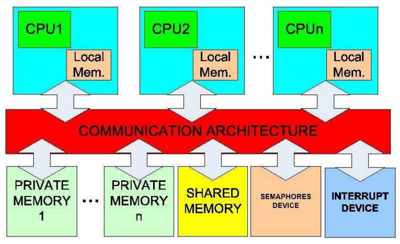


Figure 6.1: Message-oriented distributed memory architecture.

core, as presented in [27], allowing efficient data transfers between the local scratchpad and non-local memories reachable through the bus. The DMA control logic supports multichannel programming, while the DMA transfer engine has a dedicated connection to the scratch-pad memory allowing fast data transfers from or to it. In order to communicate each others, cores use non-cachable shared memory.

For the synchronization among the processors, semaphore and interrupt facilities are used:

1. a core can send interrupt signals to each other using the hardware interrupt module mapped on in the global addressing space;

2. several cores can synchronize using the semaphore module that implements test-and-set operations.

Finally, each processor core has a private on-chip memory, which can be accessed only by gaining bus ownership. In principle, it could be also an off-chip memory. In any case, it has a higher access cost and can be used to store program operands that do not fit in scratch-pad memory. Optimal memory allocation of task program data to the scratch-pad versus the private memory is a specific goal of our optimization framework, dealing with the constraint of limited size of local memories in on-chip multi-processors.

The software support is provided by a real-time operating system called RTEMS [11].

Our implementation thus supports:

- either processor or DMA-initiated memory-to-memory transfers,

- either polling-based or interrupt-based synchronization, and

- flexible allocation of the consumer's message buffer to the local scratch-pad or the non-local private memory.

The architecture is assumed to provide a hardware-software support for messaging, targeting scalability to a large number of communicating cores. Messages can be exchanged by tasks through software communication queues, which can be physically allocated either in scratch-pad memory or in shared memory, depending on whether tasks are mapped onto the same processor or not. This assumption avoids to generate bus traffic and to incur congestion delays for local communications. We also target architectures where synchronization between producer-consumer pairs does not give rise to semaphore polling traffic on the bus, since this might unacceptably and unpredictably degrade performance of ongoing message exchanges. Interrupt-based synchronization or the implementation of distributed semaphores at each computation tile are two example mechanisms matching our requirements.

6.5 High-Level Application

The multi-task application to be mapped and executed on top of the target hardware platform is represented as a conditional task graph with precedence constraints. In the following we describe some preliminaries on Conditional Task Graph and on the high level application.

6.5.1 Conditional Task Graph

A CTG is a tuple $\langle T, A, C, P\rangle$, where

- $T = T_B \cup T_F$ is a set of nodes; $t_i \in T_B$ is called branch node, while $t_i \in T_F$ is a fork node.
- A is a set of arcs as ordered pairs $a_k = (t_i, t_j)$.
- C is a set of pairs $\langle t_k, c_k\rangle$ for each node $t_k \in T_B$ representing the condition labeling the node.
- P is a set of triples $\langle Arc, Out, Prob\rangle$ each one labeling an arc $Arc = (t_k, t_j)$ rooted in a branch node t_k, Out is a possible outcome of condition c_k labeling node t_k, and p_k is the probability that Out is true ($p_k \in [0, 1]$).

The CTG always contains a single root node with no incoming arcs.

Intuitively fork nodes originate parallel activities, while branch nodes have mutually exclusive outgoing arcs. We also need to define and-nodes and or-nodes. A node with more than one ingoing arc is an *or-node* if all ingoing arcs are mutually exclusive, it is instead an *and-node* if all arcs are not mutually exclusive; mixed nodes are not allowed.

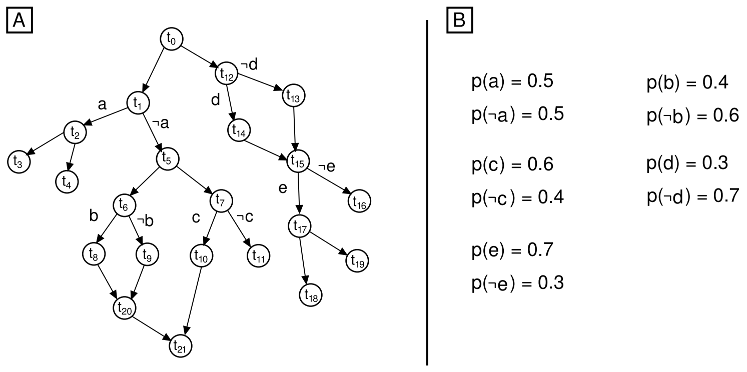


Figure 6.2: A Conditional Task Graph

For instance, in figure 6.2A t_0 is the root node and it is a fork node. Arcs (t_0, t_1) and (t_0, t_{12}) rooted in a fork node are deterministic. Node t_1 is a branch node, labeled with condition a. With an abuse of notation we have omitted the condition in the node and we have labeled arc (t_1, t_2) with a meaning $a = true$ and (t_1, t_5) with $\neg a$ meaning $a = false$. The probability of $a = true$ is 0.5 and the probability of $a = false$ is also 0.5 as depicted in figure 6.2B. Node t_{20} is an or-node while node t_{21} is an and-node.

Since the truth or the falsity of conditions is not known in advance, the model is stochastic. For the purposes of this chapter, we are interested in the concept of scenario. A scenario corresponds to an assignment of outcomes to conditions and defines a deterministic task graph containing the set of nodes and arcs that are active in the scenario. In figure 6.2 an example of run time scenario is defined by the assignment $a = true, d = true$ and $e = false$.

Given a CTG=$\langle T, A, C, P\rangle$, and a scenario s, the deterministic task graph TG(s) associated with s is defined as follows:

- The root node always belongs to the TG(s)
- An arc (t_i, t_j) belongs to TG(s) if it is a deterministic arc and t_i belongs to TG(s) or if it has an associated outcome $Out_{ij} \in s$ and t_i belongs to TG(s).
- A node t_i belongs to TG(s) if it is an and-node and all arcs $a_k \in A^-(t_i)$ are in TG(s) or if it is an or-node and only one arc $a_k \in A^-(t_i)$ is in TG(s)

The deterministic task graph derived from the CTG in figure 6.2 and associated to the run time scenario $a = true$, $d = true$ and $e = false$ is depicted in figure 6.3.

We need now to associate a probability to each scenario.

$$\forall s \in S \;\; p(s) = \prod_{Out_{ij} \in s} p_{ij}$$

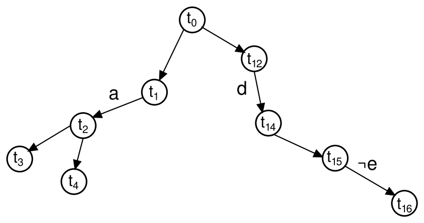


Figure 6.3: The deterministic task graph associated to the run time scenario $a = true$, $d = true$ and $e = false$

In this chapter we are interested in computing the probability of sets of scenarios. In particular, we are interested in all scenarios where a given task is active (executes) and in all scenarios where pairs of tasks are active. The probability of the scenarios where task t_i is active can be computed as follows:

$$p(t_i) = \sum_{s | t_i \in TG(s)} p(s)$$

while the probability a couple of tasks are both active in the same scenario is

$$p(t_i \wedge t_j) = \sum_{s | t_i, t_j \in TG(s)} p(s)$$

Finally, for modeling purposes, we also define for each task an activation function $f_{t_i}(s)$; this is a stochastic function such that:

$$f_{t_i} : S \rightarrow \{0, 1\}$$

and

$$f_{t_i}(s) = \begin{cases} 1 & \text{if } t_i \in TG(s) \\ 0 & \text{otherwise} \end{cases}$$

6.5.2 Application Model

Given a CTG representing the high level application, we interpret each node as a task and each arc as a communication between two tasks.

Computation, storage and communication requirements are annotated onto the graph. In detail, the worst case execution time (WCET) is specified for each node/task and plays a critical role whenever application real-time constraints (expressed here in terms of deadlines) are to be met.

Each node/task t_i also has two kinds of associated memory requirements:

- **Program Data**: storage locations are required for computation data and for processor instructions; we refer to this quantity as m_i.

- **Internal State**: a task internal state can be stored either locally or remotely; we refer to this quantity as st_i

Each arc between two tasks $r = (t_i, t_j)$ is labelled with the memory requirement for **Communication queues**.

Each of these memory requirement can be allocated either locally in the scratchpad memory or remotely in the on-chip memory; a local allocation of internal state and program data is allowed on the processor where the corresponding task runs. The Communication queue related to arc (t_i, t_j) can be allocated locally only if both t_i and t_j run on the same processor. The communication requirement, i.e., the amount of data that need to be exchanged between two tasks is referred to as c_r.

6.6 Problem model

The problem we face is the following: given a CTG, we have to map each node/task of the CTG onto a processing element, and each memory requirements (program data, internal state and communication queues) on a local/remote storage device, and to schedule tasks and communications on the available resources. Since the problem is stochastic we have to guarantee that for each possible run time scenario all temporal and resource constraints are satisfied. The objective function we have to minimize is the bus usage. Being the problem stochastic, we should minimize the expected bus utilization instead of its real value on a specific scenario. Therefore the optimal solution to our problem is a unique assignment of starting times and resources to tasks that is feasible whatever the run time scenario is, that minimizes the expected value of the bus utilization.

Note that the bus utilization to be minimized counts two contributions: one related to single tasks, since once computation data and/or internal state are physically allocated to remote memory a number of bus accesses should be performed. This communication depends on the amount of data to be stored. The second contribution is related to pairs of communicating tasks in the task graph. If two communicating tasks are allocated onto two different processors they should access the bus. This contribution depends on the amount of data the two tasks should exchange.

6.6.1 Problem structure

The problem considered is a scheduling problem with alternative resources. In fact, each task should be allocated to a processor. Each memory slot required for processing the task should be allocated to a memory device. Clearly, tasks should be scheduled in time subject to real time constraints, precedence constraints, and resource capacity constraints.

From a different perspective, the problem decomposes into two components:

- the allocation of tasks to processors and of the memory slots required by each task to the proper memory device;

- a scheduling problem with static resource allocation.

The objective function of the overall problem is the minimization of the expected communication cost. This function involves only variables of the first problem.

A number of papers in the recent literature [8], [9], [1], [31] suggest that these kinds of problems are efficiently solved via the so called Logic-based Benders Decomposition that works as follows: the allocation problem (called master problem) is solved first, and the scheduling problem (called subproblem) later. The master is solved to optimality and its solution passed to the subproblem solver. If the solution of the master is feasible for the subproblem constraints, then the overall problem is solved to optimality. If, instead, the master solution cannot be completed by the subproblem solver, a no-good is generated and added to the model of the master problem, roughly stating that the solution passed should not be recomputed again (it becomes infeasible), and a new optimal solution is found for the master problem respecting the (set of) no-good(s) generated so far.

Given the structure of the allocation and scheduling problems, we have solved the allocation problem via Integer Linear Programming and the scheduling problem via Constraint Programming.

Constraint Programming (CP) has been recognized as a suitable modelling and solving tool to face combinatorial (optimization) problems. Problems are modeled declaratively by defining a set of variables representing problem entities, each variable has an associated domain representing possible variable assignments and a set of constraints, limiting the values that variables can simultaneously assume. A solution of a constraint program is an assignment of values to variables which is consistent with constraints. The solving process of a constraint solver is the following. Each constraint is propagated so as to remove a priori those values that cannot appear in any consistent solution. Then, since propagation is not complete, i.e., some values left in the domain can still be inconsistent, tree search is performed. The process of constraint propagation and search is iterated as long as a solution is found or a failure occurs. One of the most successful application of CP to date is scheduling. Problem variables are activity starting times. Temporal constraints are easily represented through mathematical constraints.

For example, if two activities act_i and act_j characterized by starting times $start(act_i)$ and $start(act_j)$ and durations d_i and d_j are linked by a precedence constraint stating that activity act_i should be executed before activity act_j, the following mathematical constraint can be imposed, $start(act_i)+d_i \leq start(act_j)$. Many resource and temporal constraints have been devised for scheduling applications. The aim of these constraints is to apply filtering algorithms that remove a priori domain values from the variable $start(act_i)$ that are inconsistent with the constraint itself. For a survey on existing constraints and filtering algorithms the interested reader can refer to [16].

Another solution technique, which is well known and widely used in the system design community is Integer Programming (IP). Integer programming is an older method, with roots that date back to the late 1950s. Integer Programming can be thought of as a restriction of Constraint Programming. In fact, Integer Programming has only two types of variables: integer variables whose domain contain non-negative integers and continuous variables whose domain contain non-negative real values. In addition, IP allows only one type of constraint: linear inequalities. Finally, the objective function must be linear in the variables.

It seems that these restrictions make Integer Programming much narrower than Constraint Programming. However, many problems can still be modeled effectively, and algorithms for integer programs can find optimal solutions quickly for many application domains.

The solving principle of IP is based on the solution of the *linear relaxation*, allowing arbitrary sets of linear constraints to be treated as a global constraint, providing a global view of the problem. The relaxation provides a bound enabling efficient pruning of the search tree and directing search toward promising regions.

6.6.2 Allocation problem model

With regards to the platform described in section 6.4, the allocation problem can be stated as the one of assigning processing elements to tasks and storage devices to their memory requirements. First, we state the stochastic allocation model, then we show how this model can be transformed into a deterministic model through the use of existence and co-existence probabilities of tasks. To compute these probabilities, we propose two polynomial time algorithms exploiting the CTG structure.

Stochastic integer linear model

Suppose n is the number of tasks, p the number of processors, and n_a the number of arcs. We introduce for each task and each PE a variable T_{ij} such that $T_{ij} = 1$ iff task i is assigned to processor j. We also define variables M_{ij} such that $M_{ij} = 1$ iff task i allocates its program data locally, $M_{ij} = 0$ otherwise. Similarly we introduce variables St_{ij} for task i internal state requirements and C_{rj} for arc r communication queue. The objective function depends also on the run time scenarios s. We call S the set of all possible run time scenarios. We want to minimize of bus traffic expected value. The allocation model is defined as follows:

$$\min z = E(\mathit{busTraffic}(M, St, C, S))$$

$$s.t. \quad \sum_{j=0}^{p-1} T_{ij} = 1 \qquad \forall i = 0, .., n-1 \tag{6.1}$$

$$St_{ij} \leq T_{ij} \qquad \forall i = 0, .., n-1, j = 0, .., p-1 \tag{6.2}$$

$$M_{ij} \leq T_{ij} \qquad \forall i = 0, .., n-1, j = 0, .., p-1 \tag{6.3}$$

$$C_{rj} \leq T_{ij} \qquad \forall arc_r = (t_i, t_k), r = 0, .., n_a-1, j = 0, .., p-1 \tag{6.4}$$

$$C_{rj} \leq T_{kj} \qquad \forall arc_r = (t_i, t_k), r = 0, .., n_a-1, j = 0, .., p-1 \tag{6.5}$$

$$\sum_{i=0}^{n-1} [st_i St_{ij} + m_i M_{ij}] + \sum_{r=0}^{n_a-1} c_r C_{rj} \leq Cap_j \qquad \forall j = 0, .., p-1 \tag{6.6}$$

Constraints (6.1) force each task to be assigned to a single processor. Constraints (6.2) and (6.3) state that program data and internal state can be locally allocated on the PE j only if task i runs on it. Constraints (6.4) and (6.5) enforce that the communication queue of arc r can be locally allocated only if both the source and the destination tasks run on processor j. Finally, constraints (6.6) ensure that the sum of locally allocated internal state (st_i), program data (m_i) and communication (c_r) memory cannot exceed the scratchpad device capacity (Cap_j). All tasks have to be considered here, regardless they execute or not at runtime, since a scratchpad memory is, by definition, statically allocated. In addition, some symmetries breaking constraints have been added to the model. All problem constraints should be verified independently from the run time scenario. On the contrary, scenarios should be considered in the objective function expected value.

The expected value of the bus traffic is computed taking into account the set S of all possible run time scenarios as follows:

$$E(\textit{busTraffic(M, St, C, S)}) = \sum_{s \in S} p(s)\textit{busTraffic(M, St, C)}(s)$$

$\textit{busTraffic}(\text{s}) = \sum_{i=0}^{n-1} \textit{taskBusTraffic}_i(s) + \sum_{arc_r=(t_i,t_k)} \textit{commBusTraffic}_r(s)$
where
$\textit{taskBusTraffic}_i(s) = f_{t_i}(s)\left[m_i(1 - \sum_{j=0}^{p-1} M_{ij}) + s_i(1 - \sum_{j=0}^{p-1} S_{ij})\right]$
$\textit{commBusTraffic}_{r=(t_i,t_k)}(s) = f_{t_i}(s)f_{t_k}(s)\left[c_r(1 - \sum_{j=0}^{p-1} C_{rj})\right]$

In the *taskBusTraffic(s)* expression, if task t_i executes in s (thus $f_{t_i}(s) = 1$), then $(1 - \sum_{j=0}^{p-1} M_{ij})$ is 1 iff the task i program data is remotely allocated. The same holds for the internal state. In the *commBusTraffic(s)* expression we have a contribution if both the source and the destination task execute ($f_{t_i}(s) = f_{t_k}(s) = 1$) and the queue is remotely allocated ($1 - \sum_{j=0}^{p-1} C_{rj} = 1$).

Transformation in a deterministic model

In most cases, the minimization of a stochastic functional, such as the expected value, is a very complex operation (even more than exponential), since it often requires to repeatedly solve a deterministic subproblem [35]. The cost of such a procedure is not affordable for hardware design purposes since the deterministic subproblem is by itself NP-hard. **One of the main contributions of this chapter is the way to reduce the bus traffic expected value to a deterministic expression.** Since all tasks have to be assigned before running the application, the allocation is a stochastic *one phase* problem: thus, for a given task-PE assignment, the expected value depends only on stochastic variables. Intuitively, if we properly weight the bus traffic contributions according to task probabilities we should be able to get an analytic expression for the expected value.

Now, since both the expected value operator and the bus traffic expression are linear, the objective function can be decomposed into task related and arc related blocks:

$$E(\textit{busTraffic}) = \sum_{s \in S} p(s)\textit{busTraffic}(s)$$

$$E(\textit{busTraffic}) = \sum_{s \in S} p(s)\left[\sum_{i=0}^{n-1} \textit{taskBusTraffic}_i(s) + \sum_{arc_r=(t_i,t_k)} \textit{commBusTraffic}_r(s)\right]$$

Since for a given allocation the objective function depends only on the stochastic variables, the contributions of decision variables are constants: we call them $KT_i = \left[m_i(1 - \sum_{j=0}^{p-1} M_{ij}) + s_i(1 - \sum_{j=0}^{p-1} S_{ij})\right]$, and $KC_r = \left[c_r(1 - \sum_{j=0}^{p-1} C_{rj})\right]$. THus

$$E(\mathit{busTraffic}) = \sum_{s \in S} p(s) \left[\sum_{i=0}^{n-1} f_{t_i}(s) KT_i + \sum_{arc_r = (t_i, t_k)} f_{t_i}(s) f_{t_k}(s) KC_r \right]$$

This can be rewritten as

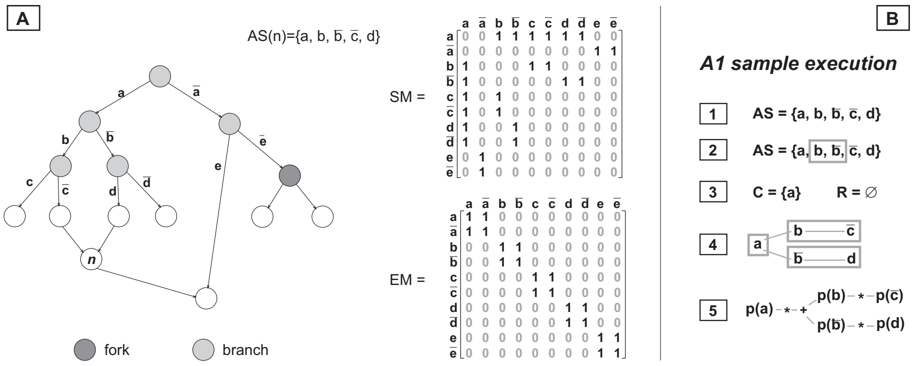
$$E(\mathit{busTraffic}) = \sum_{i=0}^{n-1} KT_i \sum_{s \in S} p(s) f_{t_i}(s) + \sum_{arc_r = (t_i, t_k)} KC_r \sum_{s \in S} p(s) f_{t_i}(s) f_{t_k}(s)$$

The term $\sum_{s \in S} p(s) f_{t_i}(s)$ is the probability of all scenarios where a given task executes (referred to as $p(t_i)$ in section 6.5.1), while $\sum_{s \in S} p(s) f_{t_i}(s) f_{t_k}(s)$ is the probability that both tasks t_i and t_k execute in the same scenario (referred to as $p(t_i \wedge t_k)$ in section 6.5.1).

$$E(\mathit{busTraffic}) = \sum_{i=0}^{n-1} KT_i p(t_i) + \sum_{arc_r = (t_i, t_k)} KC_r p(t_i \wedge t_k) \qquad (6.7)$$

To apply the transformation we need both those probabilities; moreover, to achieve an effective overall complexity reduction, they have to be computed in a reasonable time. We developed two polynomial cost algorithms to compute these probabilities.

Probability of a node



$$SM = \begin{array}{c|cccccccccc} & a & \bar{a} & b & \bar{b} & c & \bar{c} & d & \bar{d} & e & \bar{e} \\ \hline a & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ \bar{a} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ b & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ \bar{b} & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ c & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \bar{c} & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ d & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \bar{d} & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ e & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \bar{e} & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

$$EM = \begin{array}{c|cccccccccc} & a & \bar{a} & b & \bar{b} & c & \bar{c} & d & \bar{d} & e & \bar{e} \\ \hline a & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \bar{a} & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ b & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ \bar{b} & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ c & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ \bar{c} & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ d & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ \bar{d} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ e & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ \bar{e} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{array}$$

Figure 6.4: A: An example of the three data structures; **B:** a sample execution of A1

All developed algorithms are based on three data structures derived from the CTG. In Figure 6.4A we show an example of a CTG on the left and the related data structures:

- the *activation set* of a node n ($AS(n)$). It is computed by traversing all paths from the starting node to n and collecting the condition outcomes on the paths. In the activation set the outcomes are not linked via logical operators. For instance the activation set of node n in figure Figure 6.4A contains outcomes `a`, `b` and `not c` from one path and `a`, `not b` and `d` from the second path. These outcomes are grouped in the activation set.

- a binary $c \times c$ *exclusion matrix (EM)* where c is the number of condition outcomes. $EM_{ij} = 1$ iff outcomes c_i and c_j are mutually exclusive (i.e. they originate at the same branch). For instance $EM_{b\bar{b}} = 1$ since they are mutually exclusive.

- a binary $c \times c$ *sequence matrix (SM)*. $SM_{ij} = 1$ iff c_i and c_j are both needed to activate some node in the CTG. For instance $SM_{a\bar{b}} = 1$ since they are both needed to activate n.

All these data structures can be extracted from the graph in polynomial time. Once they are available, we can determine the existence probability of a node i using algorithm A1, that is used to compute $p(t_i)$ in equation (6.7). The algorithm has $O(c^3)$ complexity (where c is the number of condition outcomes) representing sets as bit vectors; in the algorithm the notation SM_i stands for the set of condition outcomes "sequenced" with a given outcome c_i ($SM_i = \{c_j | SM_{ij} = 1\}$); the same holds for EM_i.

algorithm: Activation set probability (A1) – probability of a node or an arc

```
1. let $S$ be the input set for this iteration; initially $S = AS(n)$
2. find a condition outcome $c_h \in S$ such that $(EM_h \setminus \{c_h\}) \cap S \neq \emptyset$
3. if such an outcome does not exist return $p = \prod_{c \in S} p(c)$
4. otherwise, set $B = EM_h \cap S$
5. compute set $C = S \cap \bigcap_{c_i \in B} SM_i$
6. compute set $R = \bigcap_{c_i \in B} (S \setminus SM_i)$
7. set $p = 0$
8. for each outcome $c_i \in B$:
    8.1. set $p = p + A1((S \cap SM_i) \setminus (C \cup R))$
9. set $p = p * A1(C) * A1(R)$
10. return $p$
end
```

Algorithm A1 works recursively partitioning the activation set of the target node: let us follow the algorithm on the example in figure 6.4B. We have to compute the probability of node n, whose activation set is `AS(n) = {a,b, not b, not c, d}`. The algorithm looks for a group of mutually exclusive condition outcomes (the B set), see `b` and `not b` in `AS(n)`. If such outcomes do not exist, the probability of the activation set S is the product of the probabilities of its elements (step 3). Otherwise, if there are at least two exclusive outcomes, the algorithm then builds a "common" set (C) and a "rest" set (R): the first contains outcomes c_j in sequence with all branch outcomes, such that $SM_{ij} = 1 \quad \forall c_i \in B$, the second outcomes c_h not in sequence, such that $SM_{ih} = 0 \quad \forall c_i \in B$. In the example `C = {a}` and `R = ∅`. Finally A1 builds for each branch outcome a set containing the sequenced outcomes ($S \cap SM_i$ at step 8.1), and chains `b` and `not c` and `not b` and `d` in figure 6.4. A1 is then recursively called on all these sets. The probabilities of sets corresponding to mutually exclusive condition outcomes are summed (step 8.1), the ones of C and R are multiplied (step 9).

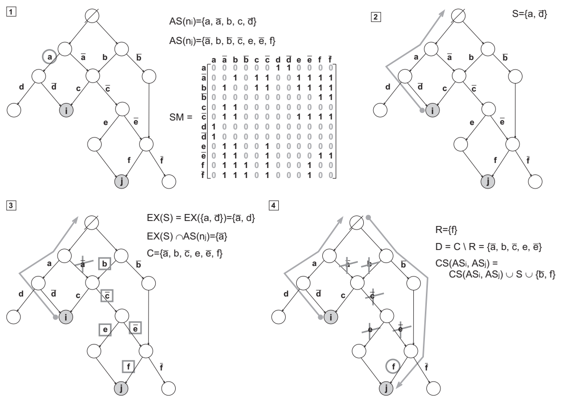
Coexistence probability of a pair of nodes

We have to compute the probability a pair of tasks are active in the same scenario so as to compute $p(t_i \wedge t_j)$ in equation (6.7). Given a pair of nodes i and j, we can determine a kind of common activation set (*coexistence set (CS)*) using algorithm A2, whose inputs are the activation sets of the two nodes ($AS(i)$, $AS(j)$). The complexity of the algorithm A2 is again $O(c^3)$. The notation $EX(S)$ stands for the exclusion set, i.e. the set of conditions surely excluded by those in S; it can be computed in $O(c^2)$.

algorithm: Coexistence set determination (A2)

```
1. if $AS_i = \emptyset$ then $CS = AS_j$; the same if $AS_j = \emptyset$
2. otherwise, if there are still not processed outcomes in
   $AS_i$, let $c_h$ be the first of them:
    2.1. compute set $S = AS_i \cap SM_h$
    2.2. compute the exclusion set $EX(S)$
    2.3. compute set:
         $C = AS_j \cap \bigcup_{c_k \in AS_j \cap EX(S)} SM_k$
    2.4. compute set:
         $R = AS_j \cap \bigcup_{c_k \in AS_j \setminus C} SM_k$
    2.5. set $D = C \setminus R$ (outcomes to delete)
    2.6. if $AS_j$ is not a subset of $D$:
        2.6.1. set $CS(AS_i, AS_j) =$
               $CS(AS_i, AS_j) \cup S \cup (AS_j \setminus D)$
end
```

A2 works trying to find all paths from n_i to the root node (backward paths) and from the root node to n_j (forward paths). The algorithm starts building a group of backward paths by choosing a condition outcome (for instance `a` in $\boxed{1}$ figure 6.5) and finding all outcomes in sequence with it (set S in $\boxed{2}$ figure 6.5).


1
AS(ni)={a, ā, b, c, d̄}
AS(nj)={ā, b, b̄, c̄, e, ē, f}
SM =
2
S={a, d̄}
3
EX(S) = EX({a, d̄})={ā, d}
EX(S) ∩AS(nj)={ā}
C={ā, b, c̄, e, ē, f}
4
R={f}
D = C \ R = {ā, b, c̄, e, ē}
CS(ASi, ASj) =
CS(ASi, ASj) ∪ S ∪ {b̄, f}

$$SM = \begin{array}{c|cccccccccccc}
 & a & \bar{a} & b & \bar{b} & c & \bar{c} & d & \bar{d} & e & \bar{e} & f & \bar{f} \\
\hline
a & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
\bar{a} & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
b & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\
\bar{b} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
c & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\bar{c} & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
d & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\bar{d} & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
e & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\bar{e} & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
f & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\
\bar{f} & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0
\end{array}$$

Figure 6.5: Coexistence set computation

Then the algorithm finds the exclusion set ($EX(S)$) of set S and intersects it with $AS(n_j)$. In $\boxed{3}$ figure 6.5 the only outcome in the intersection is `not a` (crossed arc): outcomes in the intersection and those sequenced with them are called "candidates outcomes" (set C in $\boxed{3}$ figure 6.5). These outcomes will be removed from $AS(n_j)$, unless they are sequenced with one or more non-candidate outcomes, i.e., they belong to the set R (for instance outcome `f` is in sequence with `not b` and is not removed from $AS(n_j)$ in $\boxed{4}$, figure 6.5). The outcomes left in $AS(n_j)$ identify a set of forward paths we are interested in. The algorithm goes on until all outcomes in $AS(n_i)$ are processed. If there is no path from n_i to n_j (i.e. the coexistence set is empty) the two nodes are mutually exclusive and their coexistence probability is 0.

The probability of a coexistence set can be computed once again by means of algorithm A1: thus, with A1 and A2 we are able to compute the existence probability of a single node and the coexistence probability of a pair of nodes. Since the algorithms complexities are polynomial, the reduction of the bus traffic to a deterministic expression can be done in polynomial time.

6.6.3 Scheduling Model

The scheduling subproblem has been solved by means of Constraint Programming. Since the objective function depends only on the allocation of tasks and memory requirements, scheduling is just a feasibility problem. Therefore we decided to provide a unique worst case schedule, forcing each task to execute after all its predecessors in any scenario. Tasks using the same resources can overlap if they can never appear in the same run time scenario (they are mutually exclusive).

Modeling tasks

Tasks have a five phase behavior (see figure 6.6): they read all communication queues, eventually read their internal state, execute, write their state and finally write all the communications queues.

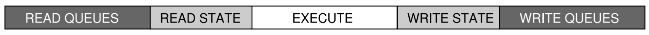


Figure 6.6: Task execution phases

For modeling purpose, each task is modeled as a set of non preemptive *activities*: in particular we use one activity to model each queue reading operation, one for each queue writing operation, one activity to represent the execution phase, one for state reading and one for state writing. Each activity has a fixed duration (DUR). The duration of execute activities is the WCET characterizing CTG nodes. For reading and writing activities (for both communication queues and state) the duration depends on the memory allocation choices made in the master; in particular local allocation of queues, state and program data results in shorter reading, writing and execution activities.

If the allocation is local to the processor where the task runs, the duration is lower than the case where the allocation of memory requirements is remote.

The activity is constrained to execute between an earliest start time (EST) and a latest end time (LET), also referred to as task deadline; for each activity *act* a *start* and an *end* variables are defined such that:

$$start(act) \geq EST(act)$$

$$end(act) \leq LET(act)$$

$$end(act) = start(act) + DUR(act)$$

Precedence relations

Tasks are linked by precedence relations due to data communication, while other precedence relations result from the decomposition of each task in many activities. Both type of relations are modeled as constraints on the start and end variables. In particular, given two activities act_i and act_j both *strict* and *loose* precedence constraint are possible, respectively enforcing $end(act_i) \leq start(act_j)$ (act_j executes after act_i) and $end(act_i) = start(act_j)$ (act_j executes immediately after act_i).

The number and type of precedence constraints used depends on the type of the involved tasks (or/and, branch/fork); an overview of all the possible schemas is given in figure 6.7. In the picture the black arrows represent strict precedence relations, while the gray hyphened arrows are loose precedence relations.

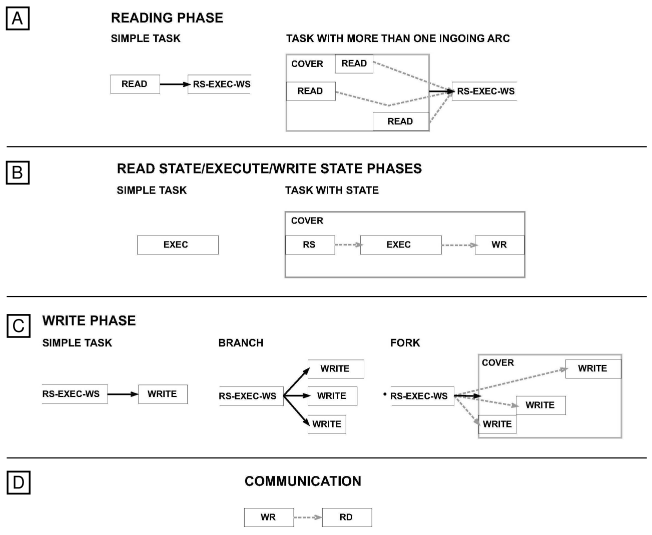


Figure 6.7: Task decomposition schema

In case a task has a single ingoing arc (input queue), the execution phase must start immediately after the only reading activity (figure 6.7A). If more than one ingoing arc is present the execution phase must start when the last reading operation is over; this is modeled by introducing a "cover" activity, which starts with the first reading activity and ends with the last one: the execution phase starts immediately after this fake activity, and loosely after each reading operation.

The execution phase consists in the only execution activity if the task has no state; otherwise the read state, execution and write state activities are linked by loose precedence relations (figure 6.7B).

If a single outgoing arc is present, the corresponding write activity must start immediately after the execution phase (see figure 6.7C). If the task has more than one outgoing arc the adopted schema depends on whether the task is branch or a fork. All the writing operations of a branch node are mutually exclusive: therefore they all start immediately after the execution phase, since they never appear in the same scenario. Writing activities of a fork node must all be performed after the execution phase, in a non specified order: a cover activity of fixed duration, equal to the sum of the durations of all writing activities, constrains their sequence to start immediately after the execution phase.

Finally, precedence relations due to data communication are modeled as loose precedence constraint between pairs of writing and reading activities corresponding to the same arc (see figure 6.7D).

Resource constraints

Both the processing elements and the bus are modeled as limited capacity resources, whose limit cannot be exceeded by overlapping non mutually exclusive tasks during the execution. On the contrary, mutually exclusive tasks can access the same resource at the same time without competition, since they never appear in the same scenario. Special resource constraints guarantee these properties to hold in the schedule.

In particular the processing elements are unary resources (i.e. with unary capacity): therefore tasks requiring the same PE cannot overlap in time at all: we modeled them defining a simple disjunctive constraint proposed in [33], which enforces, for every two activities act_i, act_j requiring the same PE:

$$p(task(act_i) \wedge task(act_j)) = 0 \vee$$

$$end(act_i) \leq start(act_j) \vee end(act_j) \leq start(act_i)$$

that is, act_i and act_j cannot overlap in time, unless they never appear in the same scenario ($p(task(act_i) \wedge task(act_j)) = 0$). In the problem we face, each activity act_i related to task t_j requires the PE t_j is mapped to. As an exception, in the execution phase of tasks with state the PE is required by a cover activity (see figure 6.7B): this allows the state reading, execution, state writing activities to stretch without releasing the processor.

The bus, as in [1], is modeled as a cumulative resource with capacity equal to its bandwidth, according with the so called "additive model", which allows an error less than 10% until bandwidth usage is under 60% of the real capacity. Each activity requires an amount of bus bandwidth dependent on the size of the data to exchange and on its duration.

A family of filtering algorithms for cumulative resource constraints is based on timetables, data structures storing the worst case usage profile of a resource over time [14]. While timetables for traditional resources are relatively simple and very efficient, computing the worst case usage profile in presence of alternative activities is not trivial, since this varies in a not linear way; furthermore, every activity can have its own resource view.

Suppose for instance we have the CTG in figure 6.8A, where for sake of simplicity each task corresponds to an activity; tasks $t_0, \ldots, t_4$ and t_6 have already been scheduled: their start time and durations are reported in figure

6.8B; the bus has bandwith 3, and the bandwidth requirement for each of the tasks is reported next to each of them in the graph. Tasks t_5 and t_7 have not yet been scheduled; t_5 is present only in scenario $\neg a$, where the bus usage profile is the first one reported in figure 6.8B; on the other hand, t_7 is present only in scenario a, b, where the bus usage profile is the latter in figure 6.8B. Therefore the resource view at a given time depends on the activity we are considering. In case an activity is present in more than one scenario, the worst case has to be considered.

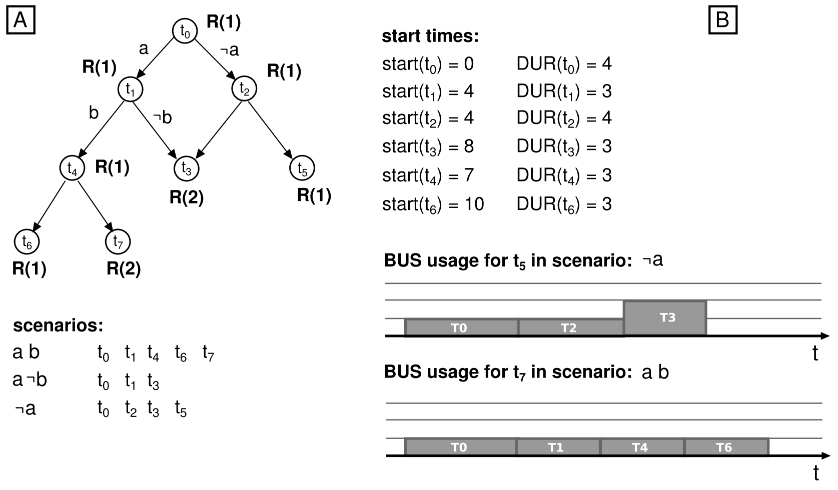

A
B
t0 R(1)
a
¬a
t1 R(1)
t2 R(1)
b
¬b
t4 R(1)
t3 R(2)
t5 R(1)
t6 R(1)
t7 R(2)
scenarios:
a b t0 t1 t4 t6 t7
a¬b t0 t1 t3
¬a t0 t2 t3 t5
start times:
start(t0) = 0 DUR(t0) = 4
start(t1) = 4 DUR(t1) = 3
start(t2) = 4 DUR(t2) = 4
start(t3) = 8 DUR(t3) = 3
start(t4) = 7 DUR(t4) = 3
start(t6) = 10 DUR(t6) = 3
BUS usage for t5 in scenario: ¬a
T0
T2
T3
t
BUS usage for t7 in scenario: a b
T0
T1
T4
T6
t

Figure 6.8: Capacity of a cumulative resource on a CTG

In order to model the bus we introduce a new global timetable constraint for cumulative resources and conditional tasks in the non preemptive case. The global constraint keeps a list of all known entry and exit points of activities: given an activity act_i, if $LST(act_i) \leq EET(act_i)$ then the entry point of act_i is $LST(t_i)$ and $EET(t_i)$ is its exit point (where LST stands for "latest start time", EET for "earliest end time" and so on).

The filtering algorithm is described in A3. Let act_i be the target activity: A3 scans the interval $[EST(t_i), finish)$ checking the resource usage at all entry points (as long as $good = true$). If it finds an entry point with not enough capacity left it starts to scan all exit points ($good = false$) in order to determine a new possible starting time for activity act_i. If such an instant is found its value is stored ($lastGoodTime$) and the finish line is updated ((step 4.2.2.2)), then A3 restarts to scan other entry points, and so on. When the finish line is reached the algorithm updates $EST(act_i)$ or fails. A3 has $O(a(c+b))$ complexity, where a is the number of activities, b the number of branches, c the number of condition outcomes. The algorithm can be easily extended to update also $LET(A)$.

algorithm: Propagation of the cumulative resource constraint with alternative activities (A3)

```
1. $time = EST(act_i), finish = EET(act_i)$
2. $latestGoodTime = time$
3. $good = true$
4. While $\neg [(good = false \wedge time > lst(a)) \vee (good = true \wedge time >= finish)]$:
    4.1. if $busreq(act_i) + usedBandwith > busBandwidth$:
        4.1.1. $time =$ next exit point
        4.1.2. $good = false$
    4.2. else:
        4.2.1. $time =$ next entry point
        4.2.2. if $good = false$:
            4.2.2.1. $lastGoodTime = time$
            4.2.2.2. $finish = max(finish, time + DUR(act_i))$
            4.2.2.3. $good = true$
5. if $good = true$: $EST(act_i) = lastGoodTime$
6. else: $fail$
end
```

A3 is able to compute the bandwidth usage seen from each activity in $O(b+c)$ by taking advantage of a particular data structure we introduced, named Branch Fork Graph (BFG).

The BFG makes it possible to compute bus usage in a very efficient way, by making direct use of the graph structure: if we only took into account the exclusion relations it would be an NP-hard problem. To have a polynomial time algorithm however the graph should satisfy a particular condition (called "Control Flow Uniqueness").

Control Flow Uniqueness

We are interested in conditional graphs satisfying *Control Flow Uniqueness* (CFU) a condition introduced in [20]. CFU is satisfied if each "and" node has a main ingoing arc arc_i, such that in every scenario where arc_i is active, also all the other ingoing arcs are active.

In practice CFU requires each and-node to be triggered by a single "main" predecessor, or, in other words, that every and-node must be on a single control path. For example in figure 6.9A, task t_5 is sufficient to trigger the execution of t_8 (since t_7 executes in all scenarios) and thus CFU holds. On the opposite, in figure 6.9B, neither t_4 nor t_5 alone are sufficient to activate t_7 and CFU is not satisfied.

In many practical cases CFU is not a restrictive assumption: for example, when the graph results from the parsing of a computer program written in a high level language (such as C++, Java, C#) CFU is naturally enforced by the scope rules of the language.

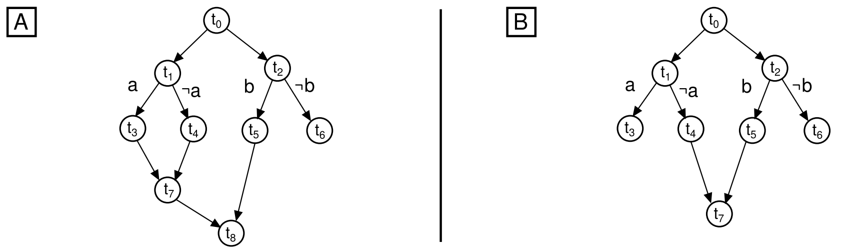


Figure 6.9: A: a CTG which satisfies CFU B: a CTG which does not satisfy CFU

In particular, CFU is a weaker condition compared to that of having structured graphs [15], that is graphs with a single collector node for each conditional branch: in particular CFU allows for graphs with multiple "tail" tasks (with no successor).

6.6.4 Benders cuts and subproblem relaxation

Each time the master problem solution is not feasible for the scheduling subproblem a cut is generated which forbids that solution. Moreover, all solutions obtained by permutation of PEs are forbidden, too.

Unfortunately, this kind of cut, although sufficient, is weak; this is why we decided to introduce another cut type, generated as follows: (1) solve to feasibility a single machine scheduling model with only one PE and tasks running on it; (2) if there is no solution the tasks considered cannot be allocated to any other PE.

The cut is very effective, but we need to solve an NP-hard problem to generate it; however, in practice, the problem can be quickly solved.

Again with the objective to limit iteration number, we also inserted in the master problem a relaxation of the subproblem. This employs two types of constraints:

1. For each sequence of communicating tasks (*path*) $\pi = t_{i_0}, t_{i_1}, \ldots$:

$$\sum_{j=0}^{n_p} \left[\sum_{t_i \in \pi} dur_i(M_{ij}, S_{ij}) + \sum_{a_r \in \pi} dur_r(E_{rj}) \right] \leq deadline$$

   That is, memory devices cannot be allocated in such a way that the total duration of a path is greater than the deadline. The linear functions $dur_i(M_i, S_i)$ and $dur_r(E_r)$ are lower bounds on the duration of all the task related activities (execution, state reading/writing) and arc related (queue reading/writing) activities on the path π.

2. For each set of non mutually exclusive tasks $S = \{t_{i_0}, t_{i_1}, \ldots\}$:

$$\forall j = 0, \ldots, n_p - 1 \quad \sum_{t_i \in S} \left[dur_{ij}(M_{tj}, S_{tj}) + \sum_{\substack{a_r = (t_i, -) \\ a_r = (-, t_i)}} dur_{rj}(E_{rj}) \right] \leq deadline$$

where $dur_{ij}(M_{tj}, S_{tj})$ and $dur_{rj}(E_{rj})$ are linear lower bounds on the duration of task t_i and arc a_r on PE j. This second type of cuts prevents the total duration of non mutually exclusive groups of activities on PE j from exceeding the deadline, since non mutually exclusive tasks cannot overlap and must execute in sequence if they are on the same PE.

Note that both the number of paths and the number of sets of non mutually exclusive tasks are exponential: relaxation cuts are therefore added during the search as they are needed.

6.6.5 Computational Efficiency

We implemented all exposed algorithms in C++, using the state of the art solvers ILOG Cplex 9.0 (for ILP) and ILOG Solver 6.0 and Scheduler 6.0 (for CP). We tested all instances on a Pentium IV pc with 512MB RAM. The time limit for the solution process was 30 minutes.

We tested the method on two set of instances: the first set contains synthetic benchmarks; peculiar input data of this problem (such as the branch probabilities) were estimated via a profiling step. Instances of this first group are only slightly structured, i.e. they have very short tracks and quite often contain singleton nodes: therefore we decided to generate a second group of instances, completely structured (one head, one tail, long tracks)[1].

The results of the tests on the first group are summarized in table 6.1. Instances are grouped according to the number of activities (acts); beside this, the table reports also the number of processing elements (PEs), the number of instances in the group (inst.), the instances which were proven to be infeasible (inf.), the mean overall time (in seconds), the mean time to analyze the graph (init), to solve the master and the subproblem, to generate the no-good cuts and the mean number of iterations (it). The solution times are of the same order of the deterministic case (scheduling of Task Graphs), which is a very good result, since we are working on conditional task graphs and thus dealing with a stochastic problem.

For a limited number of instances the overall solving time was exceptionally high: the last column in the table shows the number of instances for which

[1]All instances are available at http://www-lia.deis.unibo.it/Staff/MichelaMilano/tests.zip

this happened, mainly due to the master problem (A), the scheduling problem (S) or the number of iterations (I). The solution time of these instances was not counted in the mean; in general it was greater than than thirty minutes.

acts	PEs	inst.	inf.	time	init	master	sub	nogood	it	A/S/I
10-12	2	6	0	0.0337	0.0208	0.0075	0.0027	0.0027	1.1667	0/0/0
13-15	2	8	1	0.5251	0.1600	0.0076	0.0040	0.0020	1.1250	0/0/0
16-18	2-3	12	0	0.1091	0.0922	0.0089	0.0067	0.0013	1.0833	0/0/0
19-21	2-3	14	1	0.1216	0.0791	0.0279	0.0079	0.0046	1.2143	0/0/0
22-24	2-3	23	4	0.2336	0.1520	0.0259	0.0061	0.0081	1.1739	0/0/0
25-27	2-3	16	3	1.7849	0.0319	1.7285	0.0108	0.0088	1.3125	0/0/0
28-30	2-3	13	2	0.3331	0.0284	0.0770	0.1900	0.0338	1.6667	0/1/0
31-33	3-4	4	2	0.3008	0.2303	0.0510	0.0040	0.0000	1.0000	0/0/0
34-36	3-4	13	4	0.6840	0.0204	0.4245	0.0132	0.0108	1.2308	0/0/0
37-39	3-4	7	0	1.5670	0.0399	1.2010	0.1384	0.1877	4.4286	0/0/0
40-42	3-4	6	3	2.9162	0.0182	0.5857	2.2267	0.0390	1.6667	0/0/0
43-45	3-4	6	1	5.3670	0.2757	4.8200	0.0630	0.2005	4.1667	0/0/0
46-48	4-5	11	0	3.2719	0.0508	0.6913	2.4616	0.0683	2.0000	1/2/0
49-51	4-5	11	1	1.9950	0.1840	1.7900	0.0071	0.0087	1.1111	1/1/0
52-54	5-6	6	0	8.0000	1.3398	1.5743	4.8788	0.2073	2.7500	1/1/0
55-67	6	8	0	2.2810	0.8333	1.4377	0.0100	0.0000	1.0000	1/4/0

Table 6.1: Results of the tests on the first group of instances (slightly structured)

acts	PEs	inst.	inf.	time	init	master	sub	nogood	it	A/S/I
20-29	2	7	2	0.5227	0.0200	0.0134	0.0090	0.0021	8.8571	0/0/0
30-39	2-3	6	0	1.7625	0.0283	1.2655	0.2057	0.2630	5.8333	0/0/0
40-49	3	3	0	0.4380	0.0313	0.3493	0.0573	0.0000	1.0000	0/0/0
50-59	3-4	7	0	1.1403	0.0310	0.6070	0.2708	0.2315	3.6667	0/0/1
60-69	4-5	4	0	10.1598	0.0385	6.8718	1.2798	1.9698	18.0000	0/0/0
70-79	4-5	4	0	88.9650	0.0428	88.6645	0.2578	0.0000	1.0000	0/0/0
80-90	4-6	7	0	202.4655	0.0755	184.0177	6.5008	11.8715	28.6667	0/0/1

Table 6.2: Result of the tests on the second group of instances (completely structured)

Although this extremely high solution time occurs with increasing frequency as the number of activities grows, it seems it is not completely determinated by that parameter: sometimes even a very small change of the deadline or of some branch probability makes the computation time explode.

We guess that in some cases, when the scheduler is the cause of inefficiency, this happens because of search heuristic: for some input graph topologies and parameter configurations the heuristic does not make the right choices and the solution time dramatically grows. Perhaps this could be avoided by randomizing the solution method and by using restart strategies [28].

The results of the second group of instances (completely structured) are reported in table 6.2. In this case the higher number of arcs (and thus of precedence constraints) reduces the time windows and makes the scheduling problem much more stable: no instance solution time exploded due to the scheduling problem. On the other hand the increased number of arcs makes the al-

		mean time to gen. a cut			
basic case:					0.0074
with relaxation based cuts (RBC):					0.0499
	number of iterations		**excution time**		
deadline	**basic case**	**with RBC**	**basic case**	**with RBC**	**result**
8557573	2	3	1.18	0.609	opt. found
625918	1	1	0.771	0.765	opt. found
590846	1	1	0.562	0.592	opt. found
473108	19	6	6.169	1.186	opt. found
464512	190	14	201.124	9.032	opt. found
454268	195	24	331.449	10.189	opt. found
444444	78	15	60.747	6.144	opt. found
433330	9	4	4.396	1.657	opt. found
430835	5	3	3.347	1.046	opt. found
430490	5	3	3.896	1.703	opt. found
427251	3	2	2.153	0.188	inf.

Table 6.3: Number of iterations without and with scheduling relaxation based cuts

location more complex and the scheduling problem approximation less strict, thus increasing the number of iterations and their duration. In two cases we go beyond the time limit.

We also ran a set of tests to verify the effectiveness of the cuts we proposed in section 6.6.4 with respect to the basic cuts removing only the solution just found: table 6.3 reports results for a 34 activities instance repeatedly solved with a decreasing deadline values, until the problem becomes infeasible. The iteration number greatly reduces. Also, despite the mean time to generate a cut grows by a factor of ten, the overall solving time per instance is definitely advantageous with the tighter cuts.

Finally, to estimate the quality of the chosen objective function (bus traffic expected value), we tested it against an easier, heuristic technique of deterministic reduction. The chosen heuristic simply optimizes bus traffic for the scenario when each branch is assigned the most likely outcome; despite its simplicity, this is a particularly relevant technique, since it is widely used in modern compilers ([26]).

We ran tests on three instances: we solved them with our method and the heuristic one (obtaining two different allocations) and we computed the bus traffic for each scenario with both the allocations. The results are shown in table 6.4, where for each instance are reported the mean, minimum and maximum quality improvement against the heuristic method. Note that on the average our method always improves the heuristic solution; moreover, our solution seems to be never much worse then the other, while it is often considerably better.

			quality improvement		
instance	**activities**	**scenarios**	**mean**	**min**	**max**
1	53	10	4.72%	-0.88%	13.08%
2	57	10	2.59%	-0.11%	8.82%
3	54	24	12.65%	-0.72%	39.22%

Table 6.4: Comparison with heuristic deterministic reduction

6.7 Efficient Application Development Support

Optmization flow requires a correspondent design-time and run-time support in the target platform matching the way the application and the architecture have been abstracted in the optimization framework and allowing the precise implementation of computed mapping solutions. In practice, such support is needed to close the abstraction gap (i.e., the deviation between the mapping problem model and the real behavior of the target platform), which is the other main objective of this chapter.

In this section we describe our new application development support. It is mainly composed by a generic Customizable Application Template and a set of high-level APIs. Our facilities tackle both OS-level issues, such as task allocation and scheduling, as well as task-level issues, like inter-task communication and synchronization. The main goal of our development framework is the exact and reliable execution of application after the optimization step, giving at the same time guarantees about high performance and constraint satisfaction.

6.7.1 Design time support: Customizable Application Template

We set up a generic customizable application template allowing software developers to easily and quickly build their parallel applications starting from a high-level task and data flow graph specification compliant to our previously described models. Programmers can at first think about their applications in terms of task dependencies and quickly draw the task graphs, and then use our tools and libraries to translate the abstract representation into C code. This way, they can devote most of their effort to the functionality of tasks rather than the implementation of their communication, synchronization and scheduling mechanisms.

More in details, users can specify the number of tasks included in the target application, their nature (e.g. branch, fork, or-node, and-node) and their precedence constraints (e.g. due to data communication), thus quickly drawing its CTG. Programmers can specify the structure of the target application by simply declaring a series of macros and data structures. Once programmer has build this application skeleton, he can focus onto the functionalities of the tasks, thus giving the main effort of his work only to the more specific and critic sections of the application.

User can configure the Customizable Application Template via XML file, which will be automaticly translated into C-code. We implemented also an Eclipse plug-in graphical interface in order to make the configuration of the Customizable Application Template easier and less error-prone. Figure.7.12 shows a snapshot of how the GUI looks like. The user can compose his application task graph simply draggin and dropping nodes (i.e. task) and arrows (i.e. precedence constraints), then our plugin will produce the XML file corresponding to its right Customizable Application Template configuration.

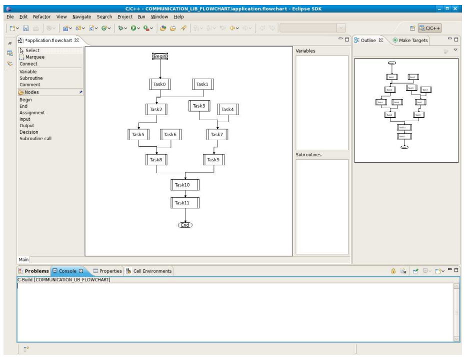


Figure 6.10: Snapshot Eclipse plug-in graphical user interface.

For every task indicated within the application template, C–code is automatically generated which reflects the considered task computational model (i.e. Reading Input Phase, Reading State Phase, Execution, etc.). Following our scalable and parameterizable template, we also ensure that the final implementation of the target application will be compliant with the modelling assumptions of the optimizer, and that the optimal performance and the constraint satisfaction of computed mapping solutions will be achieved in practice.

6.7.2 Run-time support: OS-level and Task-level APIs

We implemented a set of APIs by which users can easily reproduce optimizer solutions on their target platform with great accuracy.

Once the target application has been implemented using our generic customizable template, tasks, program data and communication queues are allocated to the proper hardware resources (processor or memory cores) as indicated by the computed allocation solution. This is done through the init task of our template which allocates and launches all the activities at booting time.

In order to reproduce the exact scheduling behavior of the optimizer, we implemented a scheduling support middleware in the target platform. Using this facility, programmers only have to specify the desired scheduling for every processor core, which is handled accordingly by our middleware in a transparent way.

After the boot of the application, our framework sets to active only the first task in the scheduling list, while the other ones are set to the sleep state. In this way, we avoid any undesired task preemption by the OS scheduler, which would induce a different behavior with respect to the optimal one provided by the optimizer.

After the active task has finished its execution, it is put to sleep thus releasing the CPU, while the subsequent task in the scheduling list is woken up by switching its state to active. If the subsequent task is allocated to a different CPU, this remote wake up mechanism is handled via interrupts.

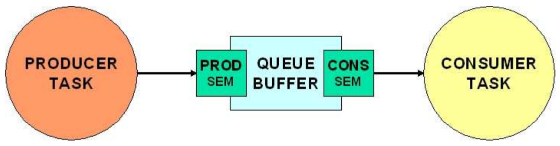


Figure 6.11: The structure of a queue.

Software support for efficient messaging is also provided by our set of high-level APIs. The communication and synchronization library abstracts low level architectural details to the programmer, such as memory maps or explicit management of hardware semaphores or interrupt signaling. Messages can be directly moved between scratch-pad memories. The structure of queue is shown in Fig. 7.17.

A queue for the communication between a producer task and a consumer one is composed by a data queue and two semaphores. In order to send a message, a producer core writes in the message queue stored in its local scratch-pad memory, without generating any traffic on the interconnect. After the message is ready, the consumer can transfer it to its own scratchpad or to a private memory space. Data can be transferred either by the processor itself or by a direct memory access controller, when available. In order to allow the consumer to read from the scratchpad memory of another processor, the scratchpad memories should be connected to the communication architecture also by means of slave ports, and their address space should be visible by the other processors.

As far as synchronization is concerned, when a producer intends to generate a message, it locally checks an integer semaphore which contains the number of free messages in the queue. If enough space is available, it decrements the semaphore and stores the message in its scratch-pad. Completion of the write transaction and availability of the message is signaled to the consumer by remotely incrementing its local semaphore. This single write operation goes through the bus. Semaphores are therefore distributed among the processing elements, resulting in two advantages: the read/write traffic to the semaphores is distributed and the producer (consumer) can locally poll whether space (a message) is available, thereby reducing bus traffic. Furthermore, our semaphores may interrupt the local processor when released, providing an alternative mechanism to polling. In fact, if the semaphore is not available, the polling task registers itself on a list of tasks waiting for that semaphore and suspends itself. Other tasks on the processor can then execute. As soon as the semaphore is released, it generates an interrupt and the corresponding interrupt routine reactivates all tasks on the wait list.

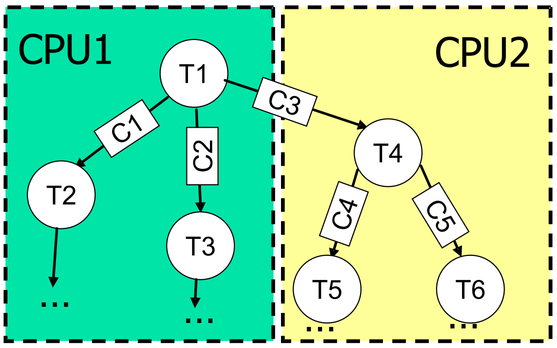


Figure 6.12: Optimal queue usage ordering: example.

If one task has got more than one input or output queue, our optimizer can specify the optimal reading/writing sequence from/to them. We tuned our run-time support to enable this option. This is a very important feature, since an optimal queue-usage ordering can boost performance and parallelism. Fig. 6.12 better clarifies this issue. It shows a case in which six tasks are allocated to two different cores. Task T1 has to communicate with both T2 and T3, which are allocated to the same core, and with T4 allocated to a different core. At start-up, let us assume that task T1 will be scheduled on CPU1 and task T4 on CPU2. While T1 immediately starts its execution, T4 has to wait for data from T1, thus keeping CPU2 stalled. The idle wait of T4 depends on the queue-fill ordering enforced by T1: it will be shorter if T1 gives maximum priority to queue C3. Both our optimization framework and our application execution support can handle this additional degree of freedom for performance optimization.

6.8 Methodology

In this section we explain how to deploy our optimization framework in the context of a real system-level design flow. Fig. 6.13 shows a pictural overview

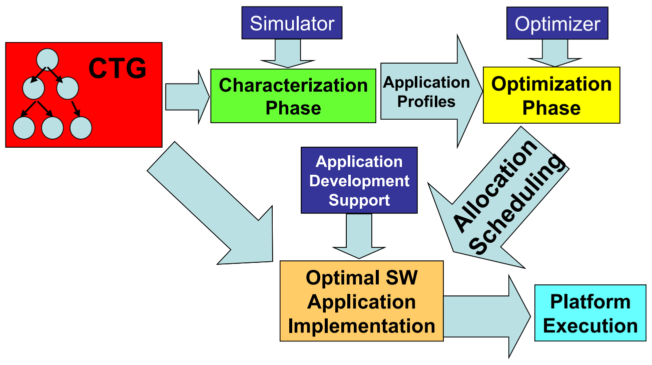


Figure 6.13: Application development methodology.

of the overall application development methodology flow proposed. Our approach consists of using a virtual platform to pre-characterize the input task set, to simulate the allocation and scheduling solutions provided by the optimizer and to detect deviations of measured performance metrics with respect to predicted ones. The target application is pre-characterized and abstracted as a Conditional Task Graph. The task graph is annotated with computation time, amount of communication and storage requirements. However, not all tasks will run on the target platform: in fact, the application contains conditional branches (like if-then-else control structures) which will prevent the execution of some of them. Therefore, an accurate application profiling step is needed, from which we have a probability distribution on each conditional branch that intuitively gives the probability of choosing that branch during real future execution.

We model task communication and computation separately to better account for their requirement on bus utilization, although from a practical viewpoint they are part of the same atomic task. The initial communication phase consumes a bus bandwidth which is determined by the hardware support for data transfer (DMA engines or not) and by the bus protocol efficiency (latency for a read transaction). The computation part of the task instead consumes an average bandwidth defined by the ratio of program data size (in case of remote mapping) and execution time. A less accurate characterization framework can be used to model the task set, though potentially incurring more uncertainty with respect to optimizer's solutions.

The input task parameters are then fed to the optimization framework, which provides optimal allocation of tasks and memory locations to processor and storage devices respectively, and a feasible schedule for the tasks meeting the real-time requirements of the application.

After the optimization phase, we can build the optimal implementation of our target application using both the optimizer solution for the hardware platform (i.e. optimal allocation and scheduling) and the application development support (i.e. Customizable Application Template and OS-level and Task-level APIs).

6.9 Experimental Results

We have performed two kinds of experiments, namely (i) comparison of simulated throughput with optimizer-derived values, and (ii) prove of viability of the proposed approach for real-life demonstrators (GSM, Software Radio).

We have deployed the virtual platform to implement the allocations and schedules generated by the optimizer, and we have measured deviations of the simulated throughput from the predicted one for 30 problem instances. A synthetic benchmark has been used for this experiment, allowing to change system and application parameters (local memory size, execution times, data size, etc.). We want to make sure that modelling approximations are not such to significantly impact the accuracy of optimizer results with respect to real-life systems.

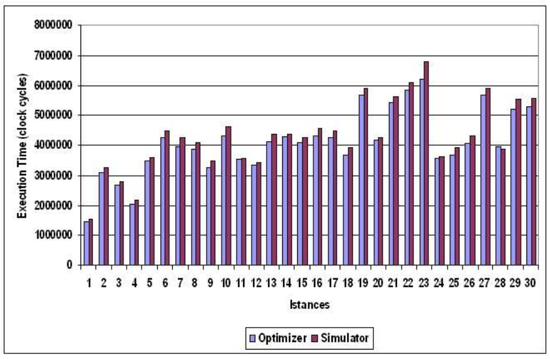


Figure 6.14: Difference in execution time

The results of the validation phase are reported in Fig.6.14 and Fig.6.15. Fig.6.14 shows the differences in execution time between the predicted one by the optimizer and the real one by the cycle accurate simulator. It can be noticed that the differences are marginal and we can point out that all the deadline constraints are satisfied.

Fig.6.15 shows the probability for throughput differences between optimizer and simulator results. The average difference between measured and predicted values is 4.8%, with 2.41 standard deviation. This confirms the high level of accuracy achieved by the developed optimization framework, thanks to the calibration of system model parameters against functional timing-accurate simulation and to the control of system working conditions.

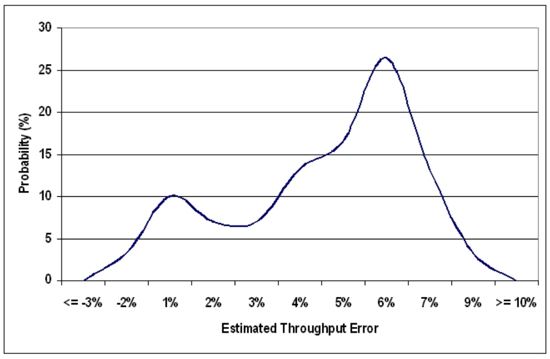


Figure 6.15: Probability for throughput differences

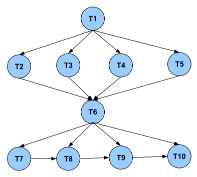


Figure 6.16: GSM encoder case study.

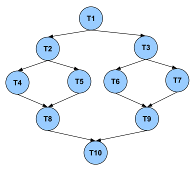


Figure 6.17: SW Radio case study.

The GSM application has been used to prove the viability of our approach. The source code has been parallelized into 10 task (see Fig.6.16), and each task has been pre-characterized by the virtual platform to provide parameters of task models to the optimizer. The time taken by the optimizer to come to a solution was 0.2 seconds. The validation process of the solution on the virtual platform running two cores showed an accuracy by 5.1% on throughput requirement.

Our optimization framework was then applied to a Software Radio application. Fig.6.17 shows the obtained task graph. The target application computation kernel was partitioned into 10 stages, and the accuracy on throughput estimation was 6.33% with a solution found in 0.25 seconds.

6.10 Conclusions

We target allocation and scheduling of conditional multi-task applications on top of distributed memory architectures with messaging support. We tackle the complexity of the problem by means of decomposition and no-good generation, and introduce a software library and API for the reliable software deployment. Moreover, we propose an entire innovative framework to help programmers in software implementation and deploy a virtual platform to validate the results of the development framework and to check modelling assumptions of optimizer, showing a very high level of accuracy. Our methodology can potentially contribute to the advance in the field of software optimization and development tools for highly integrated on-chip multiprocessors.

Bibliography

[1] M. Horowitz Scaling, Power and the Future of CMOS *VLSID '07: Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference*, 2007, IEEE Computer Society.

[2] R. W. Brodersen and M. A. Horowitz and D. Markovic and B. Nikolic and V. Stojanovic Methods for true power minimization *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, 2002, 35–42, ACM.

[3] T. Mudge Power: A First-Class Architectural Design Constraint *Computer*, vol.34, n.4, 2001, 52–58, IEEE Computer Society Press

[4] M. Horowitz et al. The Future of Wires *Proc. IEEE*, vol.89, 2001, 490–504.

[5] S. Borkar. Design Challenges of Technology Scaling *IEEE Micro*, vol.19, n.4, 1999, 23–29,IEEE Computer Society Press.

[6] S. Borkar. Thousand core chips: a technology perspective *DAC '07: Proceedings of the 44th annual conference on Design automation*, 2007, 746–749.

[7] J. R. Graham. Integrating parallel programming techniques into traditional computer science curricula *In: SIGCSE Bull.*, vol.39, n.4, 2007, 75–78, ACM.

[8] G. Martin. Overview of the MPSoC design challenge *In: DAC '06: Proceedings of the 43rd annual conference on Design automation*, 2006, 274–279.

[9] S. Medardoni and M. Ruggiero and D. Bertozzi and L. Benini and G. Strano and C. Pistritto Interactive presentation: Capturing the interaction of the communication, memory and I/O subsystems in memory-centric industrial MPSoC platforms *In: DATE '07: Proceedings of the conference on Design, automation and test in Europe*, 2007

[10] A. Andrei and M. Schmitz and P. Eles and Z. Peng and B. M. Al-Hashimi. Overhead-Conscious Voltage Selection for Dynamic and Leakage Energy

Reduction of Time-Constrained Systems *In: DATE '04: Proceedings of the conference on Design, automation and test in Europe*, 2004

[11] Rtems home page, http://www.rtems.com.

[12] S. Ahmed and A. Shapiro. The sample average approximation method for stochastic programs with integer recourse. *In: Optimization on line.*, 2002.

[13] J. Axelsson. Architecture synthesis and partitioning of real-time systems: A comparison of three heuristic search strategies. In *CODES '97*.

[14] P. Laborie. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results In *Artif. Intell. 143(2): 151-188 (2003)*.

[15] Y. Xie, W. Wolf. Allocation and scheduling of conditional task graph in hardware/software co-synthesis. In *DATE 2001: 620-625*.

[16] P. Baptiste, C. Le Pape, W. Nuijten, Constraint-Based Scheduling Kluwer Academic Publisher, 2003.

[17] A. Bender. Milp based task mapping for heterogeneous multiprocessor systems. In *EURO-DAC '96/EURO-VHDL '96*.

[18] J. Benders. Partitioning procedures for solving mixed-variables programming problems. *Computational Management Science*, 2005.

[19] L. Benini, D. Bertozzi, A. Guerri, and M. Milano. Allocation and scheduling for MPSoCs via decomposition and no-good generation. In *Proc. of the Int.l Conference on Principles and Practice of Constraint Programming. (2005)*.

[20] M. Lombardi and M. Milano. Stochastic Allocation and Scheduling for Conditional Task Graphs in MPSoCs. In *Proc. of the Int.l Conference on Principles and Practice of Constraint Programming. (2006)*.

[21] B.Flachs. A streaming processor unit for the cell processor. In *pp.134-135, ISSCC 2005.*

[22] K. S. Chatha and R. Vemuri. Hardware-software partitioning and pipelined scheduling of transformative applications. *IEEE Trans. Very Large Scale Integr. Syst.*

[23] P. Eles, K. Kuchcinski, Z. Peng, A. Doboli, and P. Pop. Scheduling of conditional process graphs for the synthesis of embedded systems. In *DATE '98*.

[24] P. Eles, Z. Peng, K. Kuchcinski, and A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. In *Journal on Design Automation for Embedded Systems, 1997.*

[25] A. Eremin and M. Wallace. Hybrid benders decomposition algorithms in constraint logic programming. In *CP '01*.

[26] P. Faraboschi, J. Fisher, and C. Young. Instruction scheduling for instruction level parallel processors. *Proceedings of the IEEE*, pages 1638–1659, 2001.

[27] P. Francesco, P. Antonio, and P. Marchal. Flexible hardware/software support for message passing on a distributed shared memory architecture. In *DATE '05*.

[28] C. Gomes, B. Selman, K. McAloon, and C. Tretkoff. Randomization in backtrack search: Exploiting heavy-tailed profiles for solving hard scheduling problems. *AIPS*, pages 208–213, 1998.

[29] J. N. Hooker. A hybrid method for planning and scheduling. In Proc. of the Int.l Conference on Principles and Practice of Constraint Programming *CP2004*.

[30] J. N. Hooker. Planning and Scheduling to Minimize Tardiness. In Proc. of the Int.l Conference on Principles and Practice of Constraint Programming *CP2005*.

[31] V. Jain and I. E. Grossmann. Algorithms for hybrid milp/cp models for a class of optimization problems. *INFORMS J. on Computing*.

[32] S. Kodase, S. Wang, Z. Gu, and K. G. Shin. Improving scalability of task allocation and scheduling in large distributed real-time systems using shared buffers. In *RTAS '03*.

[33] K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Trans. Des. Autom. Electron. Syst.*

[34] K. Kuchcinski. Embedded system synthesis by timing constraint solving. In *IEEE Transactions on CAD*.

[35] G. Laporte and F. Louveaux. The integer l-shaped method for stochastic integer programs with complete recourse. *In: Operations Research Letters*, 1993.

[36] M. Lombardi and M. Milano. Stochastic allocation and scheduling for conditional task graphs in mpsocs. In *Technical Report 77 LIA-003-06*.

[37] V. I. Norkin, G. C. Pflug, and A. Ruszczy&#324;ski. A branch and bound method for stochastic global optimization. *Math. Program.*, 1998.

[38] P. Palazzari, L. Baldini, and M. Coli. Synthesis of pipelined systems for the contemporaneous execution of periodic and aperiodic tasks with hard real-time constraints. *ipdps*.

[39] D. Pham and et al. The design and implementation of a first-generation cell processor.

[40] S. Prakash and A. C. Parker. Sos: synthesis of application-specific heterogeneous multiprocessor systems.

[41] M. Ruggiero, A. Guerri, D. Bertozzi, F. Poletti, and M. Milano. Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip. In *DATE '06*.

[42] A. Semiconductor. Arm11 mpcore multiprocessor. In *http://arm.convergencepromotions.com/catalog/753.htm*.

[43] R. Szymanek and K. Kuchcinski. A constructive algorithm for memory-aware task assignment and scheduling. In *CODES '01*.

[44] S. A. Tarim, S. Manandhar, and T. Walsh. Stochastic constraint programming: A scenario-based approach. *Constraints, 2006*.

[45] C. Technologies. The multi-core dsp advantage for multimedia. In *Available at http://www.cradle.com/.*

[46] E. S. Thorsteinsson. Branch-and-check: A hybrid framework integrating mixed integer programming and constraint logic programming. In *CP '01*.

[47] T. Walsh. Stochastic constraint programming. *Proc. of ECAI*, 2002.

[48] D. Wu, B. Al-Hashimi, and P. Eles. Scheduling and mapping of conditional task graph for the synthesis of low power embedded systems. In *Computers and Digital Techniques, IEE Proceedings. Volume 150 (5). (2003)*.

[49] Intel Corp. Intel IXP2800 Network Processor Product Brief, 2002.

[50] http://multicore.amd.com/us-en/AMD-Multi-Core.aspx.

[51] http://www.st.com/stonline/products/families/mobile/processors/processorsprod.htm.

[52] http://www.cisco.com/en/US/products/ps5763/.

[53] http://www.nec.co.jp/techrep/en/journal/g06/n03/060311.html

Chapter 7

Cellflow: a Parallel Application Development Infrastructure with Run-Time Support for the Cell BE Processor

7.1 Overview

The Cell BE processor provides both scalable computation power and flexibility, and it is already being adopted for many computational intensive applications like aerospace, defense, medical imaging and gaming. Despite of its merits, it also presents many challenges, as it is now widely known that is very difficult to program the Cell BE in an efficient manner. Hence, the creation of an efficient software development framework is becoming the key challenge for this computational platform.

We propose a novel software toolkit (called Cellflow) which enables developers to quickly build multi-task applications for Cell-based platform. We support programmers from the initial stage of their work, through a development-time software infrastructure, to the final stage of the application development, proposing a safe and easy-to-use explicit parallel programming model.

We address also the problem of allocating and scheduling of tasks on processor engines, as well as communication channels to memories, with the goal of minimizing application execution time. We have developed a complete optimization strategy based on problem decomposition. Unfortunately, a traditional two-stage decomposition produces unbalanced components: allocation part is difficult, while the scheduling part is much easier. To address this issue, we have developed a multi-stage decomposition, which is a recursive application of standard Logic based Benders' Decomposition (LBD). Our experiments demonstrate that this apprach is very effective in obtaining balanced sub-problems and in reducing the runtime of the optimizer. Our environment provides also on-line runtime support, which manages OS-level issues (such as task allocation and scheduling) as well as task-level issues (like inter-task communication and synchronization).

Experimental results show that in Cellflow we reduced to minimum the abstraction gap between the optimization and development phases.

7.2 Introduction

Cell is a heterogeneous multi-core architecture composed by a standard general purpose microprocessor (called PPE), with eight coprocessing units (called SPEs) integrated on the same chip. The SPE is a processor designed for streaming workloads, featuring a local memory, and a globally-coherent DMA engine [21], [26]. Cell has already demonstrated impressive performance ratings in computationally intensive applications and kernels mainly thanks to its innovative architectural features [34], [19], [33], [30]. Unfortunately, Cell's main differences from conventional homogeneous multiprocessors are at the same time the reason for its programming difficulties. The heterogeneity of its computational capability, the limited, explicitly-managed on-chip memory and the multiple options for exploiting hardware parallelism, make efficient application design and implementation a major challenge. Efficiently programming requires to explicitly manage the resources available to each SPE, as well the allocation and scheduling of activities on them, the storage resources, the movement of data and synchronizations, etc.

Even though data-flow graphs are often used for pure data-streaming applications, many realistic applications can only be specified as generic task graphs. The problem of allocating and scheduling generic task graphs on processor engines in a distributed real-time system is NP-hard.

Moving from these considerations, the novelty of this work is the creation of a framework, called Cellflow, that can help programmers in handling these complex and critical activities and decisions. The Cell's SDK from IBM [18] is a complete and very powerful environment, but it does not offer any facility for optimizing the resource utilization in terms of both allocation and scheduling, memory transfers and utilization. Our goal is to enable developers to quickly build multi-task applications using an explicit parallel programming model. Our key object is to give developers access to the power of Cell multi-core architecture, but at a high level. We want to set programmers free from the issue of managing low-level and architecture-specific details, so they can focus on developing the core algorithms of the application. Our entire infrastructure is built on top of the low-level libraries available within the Cell's Software Development Kit (SDK) and can be fully integrated in it.

In optimization tools many simplifying assumptions are generally considered: model simplification is often achieved by abstracting away platform implementation details such as the limited capacity of local memories or the actual paradigm of communication between cores. As a result optmization problems become more tractable and easy to solve. Unfortunatly, this approach creates an abstraction gap between the optimization model and the real HW/SW platform. The abstraction gap between high level optimization tools and standard application programming models can introduce unpredictable and undesired behaviours in the final platform implementation. In the application developing phase, programmers must be conscious about system simplifications taken into account in optimization tools. For instance, a communication or synchronization sub-optimal task implementation leads to reduced throughput and/or increase latency. The main goal of our work is to address the abstraction gap, formulating a very accurate model for allocation and scheduling, which accounts for a number of non idealities in real-life hardware platforms and which is behavioural compliant with our application modelling.

Our toolkit is made of an off-line development framework and an on-line runtime support. The off-line facility is a design-time software infrastructure for the deployment of multi-task applications. It is made up of a generic customizable application template, thanks to which software developers can easily and quickly build their application skeleton starting from a high level task and data flow graph, and of an allocation and scheduling support, in order to find an optimal mapping and scheduling on the hardware architecture.

We modeled the application as a task graph. The application workload is partitioned into computation sub-units denoted as tasks, which are the nodes of the graph. Graph edges connecting any two nodes indicate task dependencies due, for example, to communication and/or synchronization. Tasks communicate through queues and each task can handle several input/output queues. We have to allocate tasks to processors, memory requirements and input/output queues to memory devices and schedule the overall application in order to minimize the application execution time (i.e., the schedule makespan). We have previously solved similar applications [1], [2] via Logic-based Benders Decomposition [7], by facing allocation via Integer Linear Programming and scheduling via Constraint Programming. In this case, a similar approach scales poorly: the two-stage decomposition produces two unbalanced components. The allocation part is extremely difficult to solve while the scheduling part is indeed easier.

We have experimented a multi-stage decomposition, which is actually a recursive application of standard Logic based Benders' Decomposition (LBD), that aims at obtaining balanced and lighter components. An extensive set of experimental results confirm that the multi-stage decomposition pays off in terms of efficiency even if for large problems where the proof of optimality cannot be completed in the available time the traditional approach provides better solutions. Also, we analyze the impact of cutting planes and number of iterations in the traditional benders approach and in the variant we propose.

The on-line runtime support is composed by a series of software libraries and APIs which manage OS-level issues, such as task allocation and scheduling, as well as task-level issues, like inter-task communication and synchronization. In this phase we tackle also the problem of the limited memory size of the SPEs, optimizing their utilization through overlaying.

The software framework is targeted towards statically-configured application, where allocation and scheduling settings are precomputed once at design time, such as many signal processing and even some multimedia applications. With Cellflow, Cell programming becomes simpler, but at the same time it achieves high efficiency thanks to the run-time support (which is tuned to the SPE harware) and to off-line optimal allocation and scheduling.

7.3 Related Work

The Cell architecture includes multiple, heterogeneous processor elements (PPE and SPEs) and Single-Instruction-Multiple-Data (SIMD) units on all SPEs. This kind of platform supports a wide range of heterogeneous parallelism levels. To our knowledge, prior work is mainly focused on trying to exploit fine grain parallelism of Cell, such as at instruction and function level, while our work is one of the few approaches at task level. In [20] authors present a framework for the automatic exploitation of the functional parallelism of a sequential program through the different SPEs. Their work is based on annotation of the source code of target application. A runtime library deals with generating threads, scheduling them on the SPEs, and transferring data to/from them. The authors in [31] present a realtime software platform for the Cell processor. It is based on the virtualization of the processing resources and a real-time resource scheduler which runs on the PPE. The compiler described in [23] implements techniques for optimizing the execution of scalar code in

SIMD units, subword optimization and other techniques. Authors in [22] describe several compiler techniques that aim at automatically generating high-quality code over a wide range of heterogeneous parallelism available on the CELL processor. Techniques include compiler-supported branch prediction, compiler-assisted instruction fetch, generation of scalar codes on SIMD units, automatic generation of SIMD codes, and data and code partitioning across the multiple SPEs in the system.

At a higher level of abstraction, [19] presents a complexity model for designing algorithms on the Cell processor, along with a systematic procedure for algorithm analysis. To estimate the execution time of the algorithm, the authors present a model which considers the computational complexity, memory access patterns, and the complexity of branching instructions. This model, coupled with the analysis procedure, should enable identification of potential implementation bottlenecks. Williams et al. [37] analyzed the performance of Cell for key scientific kernels such as dense matrix multiply, sparse matrix vector multiply and 1D and 2D fast Fourier transforms, while the paper [36] evaluates the performance of bioinformatic applications on the Cell. Authors in [32] provide a software development platform which allows to use standard C++ programming to create parallel applications, or extend existing applications to run on Cell. The work in [27] analyzes the performance and the available bandwidth of Cell processor, its interconnect bus and memory hierarchy for high memory bandwidth applications. The authors draw many interesting conclusions, including the statement that individual SPE to SPE communication almost achieves the peak bandwidth. Some parallel programming models have been implemented and ported on the Cell processor [29, 38]. The authors in [38] have ported Streamit and its run-time environment on Cell architecture. Streamit is based on a dataflow programming language, but it needs its own compiler, while in our case we are fully compatible with the standard C-based development flow.

7.4 Cell BE Hardware Architecture

In this section we give a brief overview of the Cell hardware architecture, focusing on the features that are most relevant for our programming enviroment. Cell is a non-homogeneous multi-core processor [35] which includes a 64-bit PowerPC processor element (PPE) and eight synergistic processor elements (SPEs), connected by an internal high bandwidth Element Interconnect Bus (EIB) [28]. Figure.7.1 shows a pictorial overview of the Cell Broadband Engine Hardware Architecture. The PPE is dedicated to the operating system and acts as the master of the system, while the eight synergistic processors

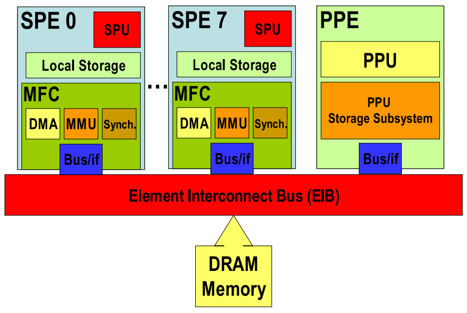


Figure 7.1: Cell Broadband Engine Hardware Architecture.

are optimized for compute-intensive applications. The PPE is a multithreaded core and has two levels of on-chip cache, however, the main computing power of the Cell processor is provided by the eight SPEs. The SPE is a compute-intensive coprocessor designed to accelerate media and streaming workloads [25]. Each SPE consists of a synergistic processor unit (SPU) and a memory flow controller (MFC). The MFC includes a DMA controller, a memory management unit (MMU), a bus interface unit, and an atomic unit for synchronization with other SPUs and the PPE.
Efficient SPE software should heavily optimize memory usage, since the SPEs operate on a limited on-chip memory (only 256 KB local store) that stores both instructions and data required by the program. The local memory of the SPEs is not coherent with the PPE main memory, and data transfers to and from the SPE local memories must be explicitly managed by using asynchronous coherent DMA commands.

7.5 Off-line Development Infrastructure

In this section, we describe the computational model supported by our environment, and the off-line (development time) support for optimal allocation and scheduling of parallel tasks on SPEs.

7.5.1 Application and task computational model

Our application model is a task graph with precedence constraints. Nodes of the graph represent concurrent tasks while the arcs indicate mutual dependencies due, for example, for communication and/or synchronization. Tasks communicate through queues and each task can handle several input/output queues.

Task execution is modeled and structured in three phases, as indicated in

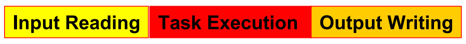


Figure 7.2: Three phases behavior of Tasks.

Figure.7.2: all input communication queues are read (Input Reading), task computation activity is performed (Task Execution) and finally all output queues are written (Output Writing). Each phase consists of an atomic activity. Each task also has 2 kinds of associated memory requirements:

1. Program Data: storage locations are required for computation data and for processor instructions;
2. Communication queues: the task needs queues to transmit and receive messages to/from other tasks, eventually mapped on different SPEs.

Both these memory requirements can be allocated on the local storage of each SPE or reside in the shared memory.

7.5.2 Multi-stage Benders Decomposition

The problem we have to solve is a scheduling problem with alternative resources and allocation dependent durations. A good way of facing these kind of problems is via Benders Decomposition, and its Logic-based extension [7]. Previous papers have shown the effectiveness of the method for similar problems. Hooker in [8] and [9] has shown how to deal with several objective functions in problems where tasks allocated on different machines are not linked by precedence constraints. Similar problems have been faced by Jain and Grossmann [6], Bockmayr and Pisaruk [4] and Sadykov and Wolsey [12], the latter comparing this approach with branch and cut and column generation. Many of these approaches consider multiple *independent* subproblems: that is, once the master problem is solved, then many decoupled subproblems result which can be solved in an independent fashion. The same approach is used by Tarim and Miguel [16] to solve stochastic problems with complete linear recourse.

The allocation is in general effectively solved through Integer Linear Programming, while scheduling is better faced via Constraint Programming. In our case, the scheduling problem cannot be divided into disjoint single machine problems since we have precedence constraints linking tasks allocated on different processors. We have implemented such an approach, similarly to [1], [2], and experimentally experienced a number of drawbacks. The main problem is that for the problem at hand a two stage decomposition produces two unbalanced components. The allocation part is extremely difficult to solve while the scheduling part is indeed easier. We will see in section 7.5.3 that this approach scales poorly.

We have experimented a multi-stage decomposition, which is actually a recursive application of standard Logic based Benders' Decomposition (LBD), that aims at obtaining balanced and lighter components. The allocation part should be decomposed again in two subproblems, each part being easily solvable.

In Figure.7.3 at level one the SPE assignment problem (SPE stage) acts as the master problem, while memory device assignment and scheduling as a whole are the subproblem. At level two (the dashed box in Figure.7.3) the memory assignment (MEM stage) is the master and the scheduling (SCHED stage) is the correspondent subproblem. The first step of the solution process is the computation of a task-to-SPE assignment; then, based on that assignment, allocation choices for all memory requirements are taken. Deciding the allocation of tasks and memory requirements univocally defines task durations. Finally, a scheduling problem with fixed resource assignments and fixed durations is solved.

When the SCHED problem is solved (no matter if a solution has been found), one or more cuts (labeled A) are generated to forbid (at least) the current memory device allocation and the process is restarted from the MEM stage; in addition, if the scheduling problem is feasible, an upper bound on the value of the next solution is also posted. When the MEM & SCHED subproblem ends (either successfully or not), more cuts (labeled B) are generated to forbid the current task-to-SPE assignment. When the SPE stage becomes infeasible the process is over converging to the optimal solution for the problem overall.

We found that quite often SPE allocation choices are by themselves very relevant: in particular, a bad SPE assignment is sometimes sufficient to make the scheduling problem infeasible. Thus, after the task to processor allocation, we can perform a first schedulability test as depicted in Figure.7.4. In practice, if the given allocation with minimal durations is already infeasible for the scheduling component, then it is useless to complete it with the memory assignment that cannot lead to any feasible solution overall.

SPE Allocation

The computation of a task-to-SPE assignment is tackled by means of Integer Linear Programming (ILP). Given a graph with n tasks, m arcs and a platform with p processing elements the ILP model we adopted is very simple: this a first visible advantage of the the multi-stage approach. We introduce a decisional variable $T_{ij} \in \{0, 1\}$ such that $T_{ij} = 1$ is task i is assigned to PE j. The model to be solved is:

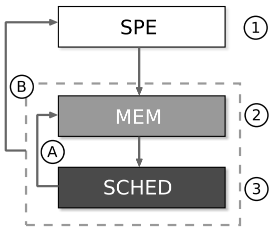


Figure 7.3: Solver architecture: two level Logic based Benders' Decomposition

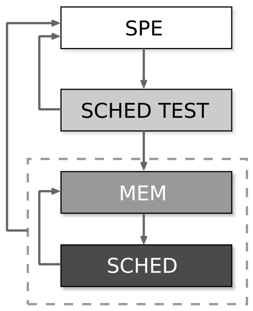


Figure 7.4: Solver architecture with schedulability test

$$\min \quad z$$

$$s.t. \quad z \geq \sum_{i=0}^{n-1} T_{ij} \qquad \forall j = 0, \ldots, p-1 \tag{7.1}$$

$$\sum_{j=0}^{p-1} T_{ij} = 1 \qquad \forall i = 0, \ldots, n-1 \tag{7.2}$$

$$T_{ij} \in \{0, 1\} \qquad \forall i = 0, \ldots, n-1, \ \forall j = 0, \ldots p-1$$

Constraints (7.2) state that each task can be assigned to a single SPE; constraints (7.1) are needed to express the objective function. The makespan objective function depends only on scheduling decision variables. Here we adopt an objective function that tends to spread tasks as much as possible on different SPEs, which often provides good makespan values pretty quickly. Constraints (7.1) force the objective variable z to be greater than the number of tasks allocated on any PE.

Constraints on the total duration of tasks on a single SPE were also added to a priori discard trivially infeasible solutions; this methodology in the LBD context is often referred to as "adding a subproblem relaxation", and is crucial for the performance of the method. In practice the model also contains the constraints:

$$\sum_{i=0}^{n-1} dmin(i) T_{ij} \leq dline \qquad \forall j = 0, \ldots, p-1$$

Where $dmin(i)$ is the minimum possible duration of task i (reading and writing phases included), and $dline$ is a deadline. Since tasks have no deadline in the present problem, we impose as deadline the makespan of the best solution found so far.

Since the SPE are symmetric resources, the allocation model also features quite standard symmetry breaking ordering constraints to remove SPE permutations.

Schedulability test

We modified the solver architecture by inserting a schedulability test between the PE and the MEM stage, as depicted in figure 7.4.

In practice, once a SPE assignment is computed, the system checks the existence of a feasible schedule using model of section 7.5.2, with all activity durations (execution, read, write) set to their minimum. If no schedule is found cuts that forbid (at least) the last SPE assignment are generated. Once a feasible schedule is found, the task-to-SPE assignment is passed to the memory device allocation component.

Memory device allocation

Once tasks are assigned to processing elements, their memory requirements and communication buffers must be properly allocated to storage devices. We tackled the problem by means of Mixed Integer Linear Programming, devising a model with a relatively simple "core".

Given a task-to-SPE assignment, for each task we introduce a boolean variable M_i such that $M_i = 1$ if t_i allocates its computation data on the local memory of the SPE it is assigned to (let this be $pe(i)$). Similarly, for each arc/communication queue $a_r = (t_h, t_k)$, we introduce two boolean variables W_r and R_r such that $W_r = 1$ if the communication buffer is on SPE $pe(h)$ (that of the producer), while $R_r = 1$ if the buffer is on SPE $pe(k)$ (that of the consumer).

$$M_i \in \{0, 1\} \qquad \forall i = 0, \ldots, n-1$$

$$W_r \in \{0, 1\}, R_r \in \{0, 1\} \qquad \forall r = 0, \ldots, m-1$$

Note that, if for an arc $a_r = (t_h, t_k)$ it holds $pe(h) \neq pe(k)$, then either the communication buffer is on the DRAM, or it is local to the producer or local to the consumer; if instead $pe(h) = pe(k)$, than the communication buffer is either on the DRAM, or it is local to both the producer and the consumer. More formally, for each arc $a_r = (t_h, t_k)$:

$$R_r + W_r \leq 1 \qquad \text{if } pe(h) \neq pe(k) \tag{7.3}$$

$$R_r = W_r \qquad \text{if } pe(h) = pe(k) \tag{7.4}$$

Constraints on the capacity of local memory devices can now be defined in terms of M, W and R variables. When a task executes it always works on local data, therefore everything it needs (input and output buffers, internal data) is copied to the local device when the task starts. At the end of the execution all data allocated in DRAM are copied back, while all locally allocated requirements are left on the local device.

Therefore, in order to state memory capacity constraint we first define:

$$base\_usage(j) = \sum_{\substack{a_r = (t_h, t_k) \\ pe(k) = j}} comm(r) R_r +$$

$$+ \sum_{pe(i)=j} mem(i) M_i +$$

$$+ \sum_{\substack{a_r = (t_h, t_k) \\ pe(h) = j \\ pe(h) \neq pe(k)}} comm(r) W_r$$

Where $mem(i)$ is the amount of memory required to store internal data of task i and $comm(r)$ is the size of the communication buffer associated to arc r. The $base\_usage(j)$ expression is the amount of memory needed to store all data permanently allocated on the local device of processor j. Then we can post the constraints:

$$\forall j = 0, \ldots, p-1, \ \forall i \text{ such that } pe(i) = j :$$

$$base\_usage(j) + \sum_{a_r = (t_h, t_i)} (1 - R_r) comm(r) +$$

$$(1 - M_i) mem(i) + \sum_{a_r = (t_i, t_k)} (1 - W_r) comm(r) \leq C_j$$

As in the previous stage, we also add to the model a scheduling subproblem relaxation; again, the two basic ideas are that the length of the longest path and the total duration of tasks on a single SPE must be lower than any current deadline. However, since memory allocation choices influence task duration, the relaxation is much more complex than that used in the SPE stage. Details on the relaxation can be found in [3].

The use of multistage Benders decomposition enables the complex resource allocation problem to be split into the drastically smaller SPE and MEM models. However, adding a decomposition step hinders the definition of high quality heuristics in the allocation stages and makes the coordination between the subproblems a critical task. We tackle these issues by devising effective Benders' cuts and using poorly informative, but very fast to optimize objective functions in the SPE and MEM stages. In practice the solver moves towards promising part of the search space by learning from its mistakes, rather than taking very good decisions in the earlier stages. Some preliminary experimental results showed how in our case this choice pays off in terms of computation time, compared to using higher quality (but harder to optimize) heuristics, or less expensive (but weaker) cuts.

Scheduling subproblem

The scheduling subproblem is modeled and solved with ILOG Scheduler. In particular, we introduce an activity for each execution phase ($exec_i$) and buffer reading/writing operation (rd_r, wr_r). Task are not preemptive, thus all activities regarding a single task execute without interruption in a pre-specified sequence. Suppose $rd_{r_0} \dots rd_{r_{h-1}}$ are the reading activities of task t_i and $wr_{r_h}, \dots, wr_{r_{k-1}}$ its writing activities, then:

$$\forall l = 0, \dots, h-2 \quad end(rd_{r_l}) = start(rd_{r_{l+1}})$$

$$end(rd_{r_{h-1}}) = start(exec_i)$$

$$end(exec_i) = start(wr_{r_h})$$

$$\forall l = h, \dots, k-2 \quad end(wr_{r_l}) = start(wr_{r_{l+1}})$$

Each communication buffer must be written before it can be read. Thus for each pair of tasks t_h, t_k linked via a precedence constraint $a_r = (t_h, t_k)$ in the task graph we impose:

$$\forall r = 0, \dots, m-1 \quad end(wr_r) \leq start(rd_r)$$

Processing elements are modeled as unary resources, and all activities regarding task t_i use SPE of index $pe(i)$. Task durations are fixed and depend on memory allocation; in particular, a local memory requirement allocation always yields smaller durations. The objective function to minimize is the makespan.

In the previous papers on similar problems [1, 15] we introduced a bus model using cumulative constraints. Here the applications we face are not communication intensive and the Cell platform provides plenty of communication bandwidth. We therefore did not impose such a constraint on the bus capacity.

Benders cuts

Benders cuts are used in the Logic Based Benders Decomposition to control the iterative solution method and are of extreme importance for the success of the approach.

In first place, cuts are generated at each iteration yielding an infeasible subproblem in order to forbid (at least) the current master problem solution; when, after a number of iterations, the master problem becomes infeasible the solution process ends. The efficiency and the effectiveness of those cuts have therefore a strong influence on the total solution time.

Second, whenever a feasible complete solution is found, a new deadline constraint is added to the makespan requiring the forthcoming solutions to be better than the current one; then, cuts for the master problem are generated as in the previous case. In principle, the effectiveness of the method could be further improved by analyzing the last feasible solution to deduce cost bounds for not yet explored master problem assignments. Unfortunately, devising effective bounds of that kind is tricky in our case, due to the presence of precedence relations between tasks on different SPEs: we therefore decided to focus on generating strong feasibility cuts.

In a multi stage Benders Decomposition approach we have to define Benders cuts for each level. Here we have to specify both level 1 and level 2 cuts: we start from the level 2 Benders cuts, between the SCHED ad the MEM stage ("A" cuts in figure 7.3).

Let σ be a solution of the MEM stage, that is an assignment of memory requirements to storage devices. If X is a variable, we denote as $\sigma(X)$ the value it takes in σ. The level 2 cuts we used are:

$$\sum_{\sigma(M_i)=0} M_i + \sum_{\sigma(R_r)=0} R_r + \sum_{\sigma(W_r)=0} W_r \geq 1 \tag{7.5}$$

This forbids the last solution σ and all solutions one can obtain from σ by remotely allocating one or more requirements previously allocated locally: this would only yield longer task durations and worse makespan. In practice we ask for at least one previously remote memory requirement to be locally allocated.

Similarly, level 1 cuts ("B" cuts in Figure.7.3), between the SPE and the MEM & SCHED stage must forbid at least the last proposed SPE assignment. Again, let σ be such a (partial) solution. Since the processing elements are symmetric resources, we can forbid together with the last assignment all its possible permutations. This is done by means of a polynomial size family of cuts.

For each processing element j we introduce a variable $S_j \in \{0, 1\}$ such that $S_j = 1$ iff all and only the tasks assigned to SPE j in σ are on a single SPE in a new solution. This is enforced by the constraints:

$$\forall j, k = 0, \dots, p-1 \qquad \sum_{\sigma(T_{ij})=1} (1 - T_{ik}) + \sum_{\sigma(T_{ij})=0} T_{ik} + S_j \geq 1 \tag{7.6}$$

We can then forbid the assignment σ and all its permutations by posting the constraint:

$$\sum_{j=0}^{p-1} S_j \leq p - 1 \tag{7.7}$$

The level 1 and level 2 cuts we have just presented are sufficient for the method to work, but they are too weak to make the solution process efficient enough; we therefore need stronger cuts. For this purpose we have devised a refinement procedure (described in Algorithm 1) aimed at identifying a subset of assignments which are responsible for the infeasibility. We apply this procedure to (7.5), (7.6) and (7.7).

Algorithm 1 Refinement procedure

```
1: let $X$ be the set of all master problem decisional variables in the original
   cut
2: sort the $X$ set in nonincreasing order according to a relevance score
3: set $lb = 0$, $ub = |X|$, $n = lb + \lfloor \frac{ub-lb}{2} \rfloor$
4: while $ub > lb$ do
5:    feed subproblem with current MP solution
6:    relax subproblem constraints linked to variables $X_{i_n}, X_{i_{n+1}}, \dots, X_{i_{|X|-1}}$
7:    solve subproblem to feasibility
8:    if $feasible$ then
9:       set $lb = n + 1$
10:   else
11:      set $ub = n$
12:   restore relaxed subproblem constraints
13: return $lb$
```

Algorithm 1 refines a cut produced for the master problem, given that the correspondent subproblem is infeasible with the current master problem solution; an example is shown in figure 7.5, where $X_{i0}, \ldots X_{i5}$ are variables involved in the Benders cut we want to refine.

First all master problem variables in the original cut (let them be in the X set) are sorted according to some relevance criterion: least relevant variables are at the end of the sequence (figure 7.5-1). The algorithm iteratively updates a lower bound (lb) and an upper bound (ub) on the number of decisional variables which are responsible for the infeasibility; initially $lb = 0$, $ub = |X|$. At each iteration an index n is computed and all subproblem constraints linked to decisional variables of index greater or equal to n are relaxed; in Figure.7.5-1 $n = 0 + \lfloor \frac{0+6}{2} \rfloor = 3$. Then, the subproblem is solved: if a feasible solution is found we know that at least variables from X_{i_0} to X_{i_n} are responsible of the infeasibility and we set the lower bound to $n+1$ (figure 7.5-2). If instead the problem is infeasible (see figure 7.5-3), we know that variables from X_{i_0} to $X_{i_{n-1}}$ are sufficient for the subproblem to be infeasible, and we can set the upper bound to n. The process stops when $lb = ub$. At that point we can restrict the original cut to variables from X_{i_0} to $X_{i_{n-1}}$.

When we apply the Algorithm 1 to level 2 cuts the X set contains all M, R and W variables in the current cut (7.5); the relevance score is the difference between the current duration of the activity they refer to in the scheduling subproblem (resp. execution, buffer reading/writing) and the minimum possible duration of the same activity. Relaxing constraints linked to M, R and W variables means to set the duration of the corresponding activities to their minimum value.

Level 1 cuts are more tricky to handle: the X set contains tasks (ranked by their minimum duration) rather than decisional variables, and to relax the constraints we have to: A) set to the minimum the duration of all activities related to the considered task; B) remove all related (7.3) and (7.4) constraints in the

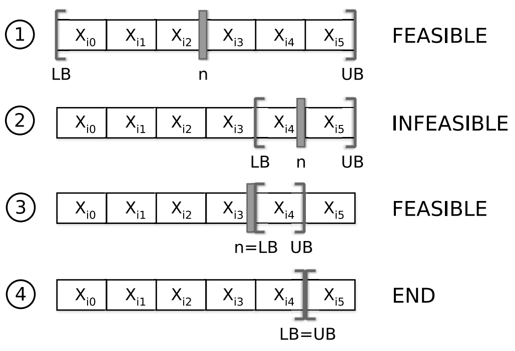

1
X_i0 X_i1 X_i2 X_i3 X_i4 X_i5
LB n UB
FEASIBLE
2
X_i0 X_i1 X_i2 X_i3 X_i4 X_i5
LB n UB
INFEASIBLE
3
X_i0 X_i1 X_i2 X_i3 X_i4 X_i5
n=LB UB
FEASIBLE
4
X_i0 X_i1 X_i2 X_i3 X_i4 X_i5
LB=UB
END

Figure 7.5: Refinement procedure: an example

memory allocation subproblem and set to 0 the memory requirement associated to all the corresponding M, R and W variables in the capacity constraints.

This cut refinement method has some analogies with what is done in Cambazard and Jussien [17], where explanations are used to generate logic based Benders cuts.

Note that refinement of level 2 cuts requires to repeatedly solve (relaxed) scheduling problems, which are by themselves NP-hard; the situation is even worse for level 1 cuts, since the subproblem is in this case the couple MEM & SCHED, which is iteratively solved. Therefore generation of refined cut is very expensive: the question is how much effort is worthwhile to spend in generating strong cuts. This is an issue which will be considered in the section about experimental results.

Finally, the described refinement procedure finds the minimum set of consecutive variables in X which cause the infeasibility of the subproblem, *without changing the order of the sequence*. Note however that is possible that some of the variables from X_{i_0} to $X_{i_{n-1}}$ are not actually necessary for the infeasibility. To overcome this limitation Algorithm 1 can be used within the iterative conflict detection algorithm described in [13], [14] to find a minimum conflict set. We implemented such an iterative procedure to generate even stronger (but of course more time consuming) cuts.

7.5.3 Computational Efficiency

Our approach has been implemented using the state of the art solvers ILOG Cplex 10.1 and Scheduler/Solver 6.3. We tested the approach on 200 task graphs representing realistic applications. All graphs were randomly generated by means of a specific instance generator designed to produce realistic task graphs. All instances feature high parallelism and complex precedence relations; durations and memory requirements are randomly generated, but based on values taken from real applications. The Cell configuration we used for the tests has 6 available SPEs.

Table 7.6 compares performance results for the traditional two stage logic based Benders decomposition approach referred to as BD, and the three stage that we propose in this chapter, referred to as TD. In the two level solver, the master problem performs allocation of tasks to SPEs and memory requirements to storage devices through Integer Linear Programming while the subproblem is a scheduling problem and is solved via Constraint Programming. Instances are grouped by number of tasks; each group contains 20 instances, for which the minimum and maximum number of arcs is also reported. The table reports the average number of SPE, MEM iterations for the three-stage approach and

		TD			BD		Timed out		
ntasks	**narcs**	**SPE it.**	**MEM it.**	**time**	**PM it.**	**time**	**TD ∧ BD**	**¬ TD ∧ BD**	**TD ∧ ¬ BD**
10-11	4-11	12	13	3.67	73	73.30	0	0	0
12-13	8-14	17	15	11.19	46	151.31	0	1	0
14-15	8-15	19	28	10.25	9	144.49	0	0	0
16-17	11-17	30	41	29.53	101	387.24	0	2	0
18-19	13-19	47	73	158.93	122	814.75	1	4	0
20-21	16-22	90	129	403.20	114	1291.90	2	10	0
22-23	19-26	87	132	571.88	95	1686.00	3	15	0
24-25	20-29	107	162	920.00	79	1639.00	9	7	0
26-27	23-29	88	187	837.50	30	1706.50	6	12	0
28-29	25-35	109	224	1218.50	24	1721.00	9	10	0

Figure 7.6: Performance tests

the average number of iterations between the master and subproblem in the two stage approach (we refer to this quantity as PM iterations). In the time columns we report the average solution time for both solvers. All tests were run with a cutoff time of 1800 seconds: the last three columns report the number of instances (out of 20) for which: 1) both TD and BD exceed the time limit (TD ∧ BD); 2) BD exceeds the time limit ad TD does not (¬ TD ∧ BD); 3) TD exceeds the time limit and BD does not (TD ∧ ¬ BD).

Note that in general TD is much more efficient than BD. Starting from group $20 - 21$, the high number of timed out instances makes the average execution time a less relevant index; by looking at the last three columns, however, one can easily see how in many large instances TD can still find the optimal solution, while BD is not able to provide it within the time limit (column ¬ TD ∧ BD); note also that the opposite never occurs (column TD ∧ ¬ BD). Of course as the number of nodes and arcs grows the number of instances for which both solvers exceed the time limit also increases (column TD ∧ BD).

Note that TD has a lower execution time, despite it generally performs more iterations than BD. This suggest that the two solvers have in practice a very different behavior: TD tends to work by solving many easy subproblems, while BD performs fewer and slower iterations.

This is more clearly shown in table 7.7, which reports for each instance group the average number of SPE, MEM, SPE & MEM (PM) and SCHED subproblems solved by both solvers. For each solver the average time to solve a single subproblem of every type is reported.

One can see how TD solves thousands of problems (mostly to generate cuts), while BD faces fewer of them. On the other hand TD subproblems are very easy; note that the difference between the number of SPE, MEM and SCHED subproblems for the TD solver is around one order of magnitude, while the time to solve each subproblem type follows an analogous, inverse trend: once again this suggest that the TD solver has a quite balanced behav-

	TD #probs			TD time			BD #probs		BD time	
ntasks	SPE	MEM	SCHED	per SPE	per MEM	per SCHED	PM	SCHED	per PM	per SCHED
10-11	12	177	484	0.0362	0.0046	0.0013	12	165	2.1314	0.0010
12-13	17	285	573	0.0954	0.0078	0.0013	13	195	4.8076	0.0014
14-15	19	389	1312	0.0291	0.0083	0.0016	14	201	6.0836	0.0016
16-17	30	692	2304	0.0656	0.0141	0.0019	18	302	35.5924	0.0017
18-19	47	1463	6014	0.1266	0.0270	0.0028	26	495	84.7409	0.0024
20-21	90	2764	12641	0.7690	0.0549	0.0030	23	428	246.3311	0.0037
22-23	83	2707	12010	0.7709	0.0585	0.0988	19	448	270.8062	0.0049
24-25	107	3807	20877	1.4909	0.0860	0.0077	10	203	773.3269	0.0055
26-27	88	3959	24692	0.6456	0.0824	0.0087	5	87	1088.9167	0.0205
28-29	109	4731	31267	1.4714	0.1091	0.0104	5	140	1080.7726	0.0099

Figure 7.7: Number of subproblems solved and their difficulty

ior. On the contrary, the resource allocation stage for the BD solver is instead often very time consuming compared to the scheduling; moreover, the gap becomes larger as the size of the instance increases.

Going more deeply, it is interesting to observe the distribution of the solution time between the problem components in the instances solved within the time limit and in those which are not.

Figure.7.8 reports histograms that show the distribution of the allocation/scheduling time ratio for the TD solver (where “allocation” means SPE + MEM). The X axis is divided into intervals, the Y axis counts the number of instances which fall in each interval.

Intuitively, in a balanced three stage decomposition strategy, the resource allocation is expected to take around 2/3 of the total solution time. One can see how the distribution for the instances solved within the time limit roughly follows a bell-shaped curve, with a peak around 0.7-0.8, slightly more than 2/3. The solution time for instances not solved within the limit appears to be more unbalanced with most of the time absorbed by the allocation. This suggests that for the TD solver more time could be spent in scheduling, for example to generate stronger cuts for the MEM stage.

This differentiated behavior between timed out and not timed out instances

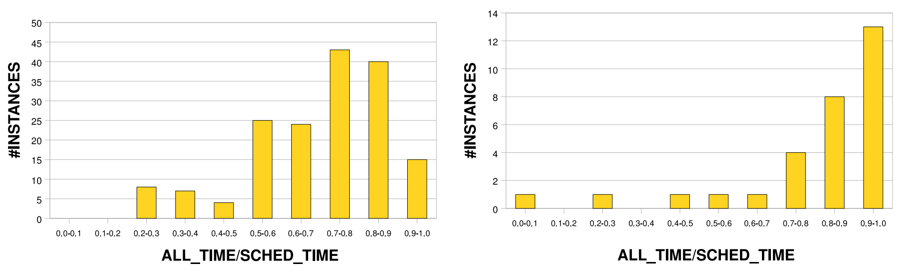


Figure 7.8: TD execution time distribution for instances solved within the time limit (on the left) and not solved within the time limit (on the right)

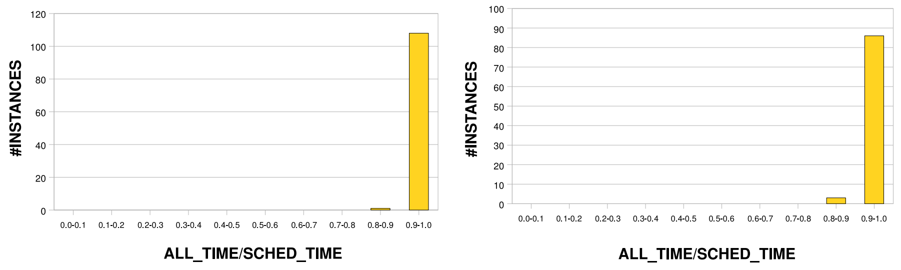


Figure 7.9: BD execution time distribution for instances solved within the time limit (on the left) and not solved within the time limit (on the right)

	Without strong ref.				With strong ref.			
ntasks	SPE it.	MEM it.	time	$> \text{TL}$	SPE it.	MEM it.	time	$> \text{TL}$
10-11	192	90	497.90	5	12	13	3.67	0
12-13	386	295	1144.21	11	17	15	11.19	0
14-15	410	539	1181.24	12	19	28	10.25	0

Figure 7.10: Performance results for the TD solver with and without strong cut refinement

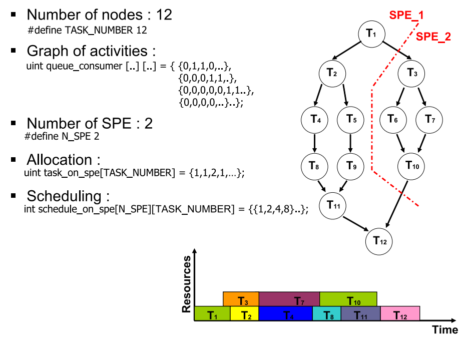
is not observed for the BD solver where substantially all the process time is spent in solving allocation subproblems (see Figure.7.9).

Since most instances in the last two groups were not solved to optimality by both the approaches, we now want to compare the solution quality when optimality is not proved. In these cases the TD solver always finds the best solution and the average improvement is around 9%.

Finally, we considered the impact of strong Benders cuts on the TD solver. We disabled the strong cut refinement system in the TD solver: instead of finding a minimum conflict at each iteration we only remove some non relevant elements, using Algorithm 1. Table 7.10 reports the number of SPE and MEM iterations, the average solution time and the number of instances not solved within the time limit for the first three groups, without and with strong cut refinement. Note how disabling the refinement process causes a drastic performance breakdown: the weak refinement procedure is therefore not strong enough. Tuning the effort to be spent in cut generation remains an open problem.

7.5.4 Customizable Application Template

We set up a generic customizable application template allowing software developers to easily and quickly build their parallel applications starting from a high-level task and data flow graph specification compliant to our previously described models. Programmers can at first think about their applications in terms of task dependencies and quickly draw the task graphs, and then use our tools and libraries to translate the abstract representation into C code. This way, they can devote most of their effort to the functionality of tasks rather than the implementation of their communication, synchronization and scheduling mechanisms.



- Number of nodes : 12

```
#define TASK_NUMBER 12
```

- Graph of activities :

```
uint queue_consumer [..] [..] = { {0,1,1,0,..},
                                  {0,0,0,1,1,.},
                                  {0,0,0,0,0,1,1..},
                                  {0,0,0,0,..}..};
```

- Number of SPE : 2

```
#define N_SPE 2
```

- Allocation :

```
uint task_on_spe[TASK_NUMBER] = {1,1,2,1,...};
```

- Scheduling :

```
int schedule_on_spe[N_SPE][TASK_NUMBER] = {{1,2,4,8}..};
```

Figure 7.11: Example of how to use the Customizable Application Template.

User can configure the Customizable Application Template via XML file, which will be automaticly translated into C-code. Figure.7.11 shows a pictorial illustration of how our C-coded template looks like.
In the example, we have depicted a task graph with twelve tasks and with precedence constraints defined in the matrix queue_consumer[][]. If task i has a precedence constraint w.r.t. task j, the element queue_consumer[i][j] will be set to 1. Information about the configuration of the target hardware platform and the desired allocation and schedule can be also specified. N_SPE macro specifies the number of available SPEs. The task_on_spe[] and schedule_on_spe[][] data structures specify where tasks should be allocated and which schedule to apply for each SPE. In a similar way, where to allocate program data and communication queues can be also defined: this is a very important feature since developer can easily and quickly find the optimal trade-off between performance and local memory occupation.

We implemented also an Eclipse plug-in graphical interface in order to make the configuration of the Customizable Application Template easier and less error-prone. Figure.7.12 shows a snapshot of how the GUI looks like. The user can compose his application task graph simply draggin and dropping nodes (i.e. task) and arrows (i.e. precedence constraints), then our plugin will produce the XML file corresponding to its right Customizable Application Template configuration.

After this configuration step, the programmer should just write the algorithms that will run on the SPEs using standard C code. Our infrastructure will automatically manage communication and synchronization between threads, exploiting at best Cell architecture features. The kernel of our customizable ap-

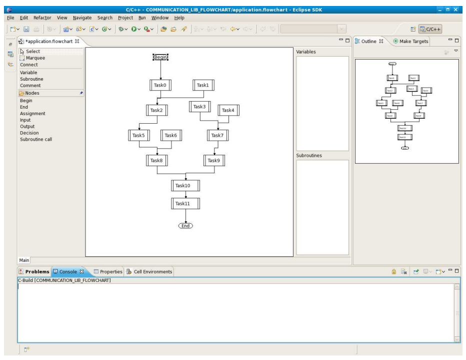
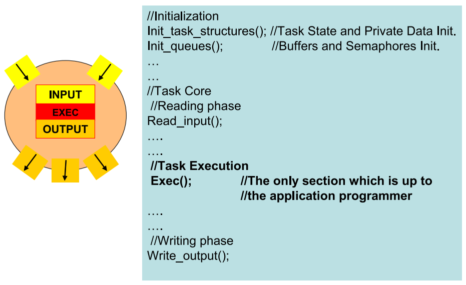


Figure 7.12: Snapshot Eclipse plug-in graphical user interface.

plication template is made of a PPE part, that reads the configuration files and sets up all the system structures, and an SPE part, that supports the run-time execution and communication. Details on how the initialization phase works and how PPE and SPE interact will be explained in section 7.6.1.



```
//Initialization
Init_task_structures(); //Task State and Private Data Init.
Init_queues();           //Buffers and Semaphores Init.
…
…
//Task Core
 //Reading phase
Read_input();
….
….
 //Task Execution
 Exec();           //The only section which is up to
                   //the application programmer
….
….
 //Writing phase
Write_output();
```

Figure 7.13: Task computational versus the generated C code.

Figure.7.13 shows C code of the SPE part. As you can notice, it reflects the considered task computational model. The INPUT phase of the computational model corresponds to the Read_input() function, while the OUTPUT phase to the Write_output() one. These two functions are blocking and handle the whole communication and synchronization procedures automatically.
The only section which has to be written by the programmer is the Exec() function: this is the customizable computational core of the task.

7.5.5 Allocation and Scheduling support

The problem of efficently allocating and scheduling multi-task applications on a multi-processor in a distributed system is very challenging. As already described in section 7.5.2 we are able to provide an optimal solution to this issue for a wide range of applications. Unfortunatly sometimes our optimization tool may exceed the time-out limit in finding the optimal solution (see section

7.5.3). To overcome this limitation, we impemented also a suboptimal algorithm based on a list scheduling heuristic [24]. We chose a heuristic algorithm because it is a good compromise between solving time and solution quality.

List scheduling keeps a list of the ready tasks (the ones whose all producers have already finished); that list is then ordered according to a priority function, and the highest-priority ready task is scheduled next. To assign priorities to tasks, ASAP (As Soon As Possible) and ALAP (As Late As Possible) start times are determined for each task, according to application task dependencies, and task mobility is calculated as the difference ALAP-ASAP. The highest mobility a task has, the highest priority it will obtain. Once the scheduling is found, the tasks are mapped on the SPEs according to a Round-Robin algorithm: proceding in priority order, each task is mapped on a different SPE. In this way it is possible to achieve a good load balancing between all the SPEs. List scheduling and round-robin (R-R) allocation are simple and scalable heuristics, but they do not provide any optimality guarantee.

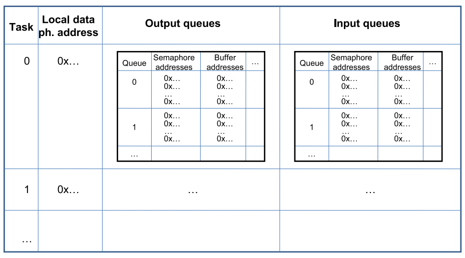
7.6 On-line Runtime Support

The runtime takes care of the task scheduling and data handling between the different cores.

7.6.1 SPE task allocator and scheduler

Once the target application has been implemented using our generic customizable template, tasks, program data and communication queues are allocated to the proper hardware resources (SPEs or memory resources). This is done through the init task of our template which allocates and launches all the activities at booting time.

More specifically, during boot the PPE creates a global configuration table that



Task	Local data ph. address	Output queues	Input queues
0	0x…	(see below)	(see below)
1	0x…	…	…
…			

Output queues / Input queues for task 0 (both printed identically):

Queue	Semaphore addresses	Buffer addresses	…
0	0x… 0x… … 0x…	0x… 0x… … 0x…	
1	0x… 0x… … 0x…	0x… 0x… … 0x…	
…			

Figure 7.14: Global application table.

contains information about queue buffers and where to allocate local data. The table is arranged so that each table entry contains information related to a task.

To populate that table, the PPE reads the configuration files (that contain allocation information) and interacts with SPEs to know the physical addresses of all structures of queues. Figure.7.14 represents a simplified global application

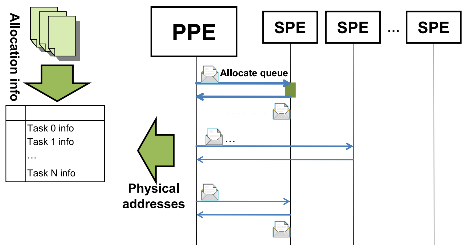


Figure 7.15: Initialization of global application table.

table. Each line is referred to a task and holds all the information that the task will need during its execution:

- local data address;
- information about input and output queues (buffer and semaphore addresses).

This interaction between PPE and SPEs uses a specific mailbox-based protocol, that supports:

- allocating local SPE data;
- allocating buffers;
- initializing semaphores;
- starting the execution (once the table is completed).

Figure.7.15 illustrates this initialization phase: the PPE gathers information from configuration files and SPEs, and builds the global application table. When a task is scheduled, its code overlay is loaded and the task's entry from the global table is received. This means a very dynamic and memory-efficient (for both code and data) local storage management to cope with its limited size. In order to reproduce the desired scheduling behavior, we implemented a scheduling support middleware. Using this facility, programmers only have to specify the desired scheduling for every SPE, which is handled accordingly by our middleware in a transparent way. To overcome the capacity limitations of local storage, we support SPE overlay: every time a new task has to be scheduled, it is loaded into local storage by our middleware through overlay. In an overlay structure the local storage is divided into a root segment, which resides always in storage, and one or more overlay regions, where overlay segments

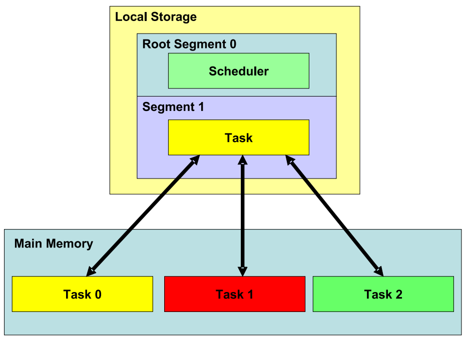


Figure 7.16: Task scheduling through overlays.

are loaded when needed. In our framework, a scheduler is implemented on each SPE and its code is stored in the root segment of the local storage (see Figure.7.16). Handling the scheduling on the SPE itself avoids additive overhead due to communication and synchronization with PPE. Our scheduling policy is non-preemptive, since the context of an SPE task is too large (it includes SPE registers, LS image, and outstanding DMA commands residing in the DMA queue) to achieve a quick context switch.

7.6.2 SPE Communication and Synchronization support

Software support for efficient messaging is also provided by our set of high-level APIs. The communication and synchronization library abstracts low level architectural details to the programmer, such as memory maps or explicit management of hardware semaphores or interrupt signaling. The structure of the queues is shown in Figure.7.17. The infrastructure for the communication be-

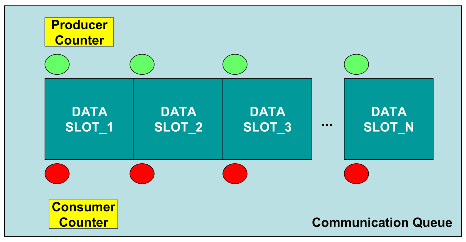


Figure 7.17: The structure of a queue.

tween a producer/consumer pair is composed by a data queue, two counters and a series of semaphores. The data queue is composed by several data slots. The data queue can be allocated either in shared memory or in local memory of SPE. A couple of semaphores is associated to each slot by means synchronization between producer/consumer pairs is implemented. Semaphores and counters are distributed and allocated in local storage to SPEs. When a producer task generates a message, it locally checks the private counter which contains the identifier of the free slot in the queue and starts to poll the slot's semaphore. When producer acquires the semaphore, it starts writing the message. If the data queue is allocated remotely (either in shared memory or in local memory to consumer) a DMA transfer is issued. When the message is ready, the producer signals this by releasing consumer's semaphore. If producer and consumer reside on different SPEs, this is the only bus access for the entire synchronization process. We set up a communication and synchronization library abstracting away low level architectural details to programmers, such as memory maps or explicit management of semaphores, DMA transfers and shared memory.

As previously mentioned, all the information about queues (i.e. structure physical addresses, ids, etc.) are stored in the task table which is filled at boot time: this brings to more efficient both communication and syncronization since the hand-shaking address negotiations are done only once and not every time a task is scheduled.

7.7 Experimental Results

7.7.1 Case study

In this section we show an example of how to use Cellflow to build a parallel application. The selected case study is an instance of a software radio application. A software radio receives its input from a data source (the digitized antenna output), while its output is connected to a digital audio output device. As Figure.7.18 shows, the main dataflow is a pipeline with a band-pass

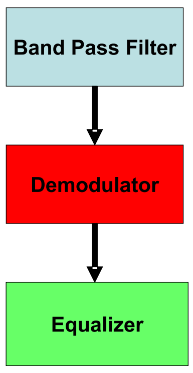
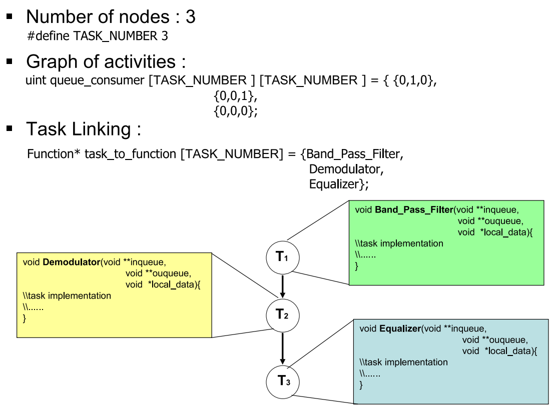


Figure 7.18: Data flow graph for a software FM radio

filter for the desired frequency, a demodulator, and an equalizer. The most intuitive and quick way to translate this data-flow into code using Cellflow is to map every node in the chart to a task. From the developer prespective, he/she will just implement the kernel code of these functional nodes and configure the application template to build the overall task graph (i.e. to specify to the run-time environment both communication and synchronization constraints between tasks). Figure.7.19 gives a simple view of the flow-graph and of the final implementation of the target software radio application.


- Number of nodes : 3

```
#define TASK_NUMBER 3
```

- Graph of activities :

```
uint queue_consumer [TASK_NUMBER ] [TASK_NUMBER ] = { {0,1,0},
                                                      {0,0,1},
                                                      {0,0,0};
```

- Task Linking :

```
Function* task_to_function [TASK_NUMBER] = {Band_Pass_Filter,
                                            Demodulator,
                                            Equalizer};
```

```
void Band_Pass_Filter(void **inqueue,
                      void **ouqueue,
                      void  *local_data){
\\task implementation
\\......
}
```

```
void Demodulator(void **inqueue,
                 void **ouqueue,
                 void  *local_data){
\\task implementation
\\......
}
```

```
void Equalizer(void **inqueue,
               void **ouqueue,
               void  *local_data){
\\task implementation
\\......
}
```

Figure 7.19: Simple dataflow graph of a software FM radio versus C code.

The above described pipeline implementation of the software radio application is the simplest way to map the data-flow chart into code. In order to increase the parallelism, the same benchmarch can be implemented splitting tasks in several sub-tasks, making the data-flow graph parallelism more explicit. More in detail, the equalizer task can be viewed as a more complex subgraph composed by different filters: it is made up of a split-join, where each child adjusts the gain over a particular frequency range, followed by a filter that adds together the outputs of each of the bands. As Figure.7.20 shows, the

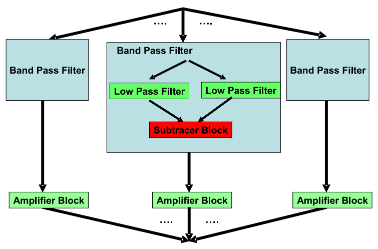


Figure 7.20: Flow graph for an equalizer.

equalizer is composed by a series of band-pass filters running in parallel, with their outputs added together. The band-pass filter can be viewed in turn as the subtraction between two low-pass filters which work at different frequency, with the overall result feeded to an amplifier. In the overall implementation through Cellflow (see Figure.7.21), the translation of this more complex dataflow graph will only reflect a different configuration of the application tem-

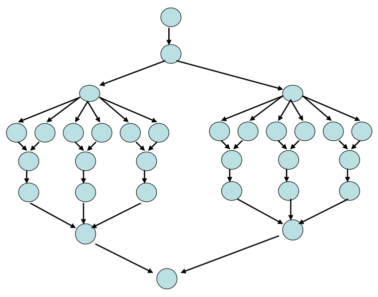


Figure 7.21: Complex data-flow graph for the Software Radio.

plate (i.e. with more tasks and a different communication and synchronization tree) and the implementation of more fine-grained kernel tasks (i.e. for the implemetation of the low-pass filter, the subtracer block and the amplifier).

7.7.2 Performance Analysis

In this section we analyze the speedup achieved by Cellflow on of three real-life applications, namely Mat-mult, FFT and Software Radio.

Mat-mult is a block matrix multiplication: each task executes a matrix multiplication between an input matrix and a private operand matrix, and then feeds its output to the following task. The platform receives a continuous flow of input matrices and produces a continuous flow of output matrices. This benchmark is representative of a wider class of applications for embedded systems with high data parallelism, like image and sound filters. The FFT benchmark is an implementation of the Fast Fourier Transform. Conceptually it is a single pipeline, but the main path is duplicated into a split-join to expose parallelism (see Figure.7.22) The Software Radio implementation has been described in section 7.7.1.

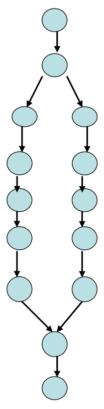


Figure 7.22: FFT-benchmark flow graph.

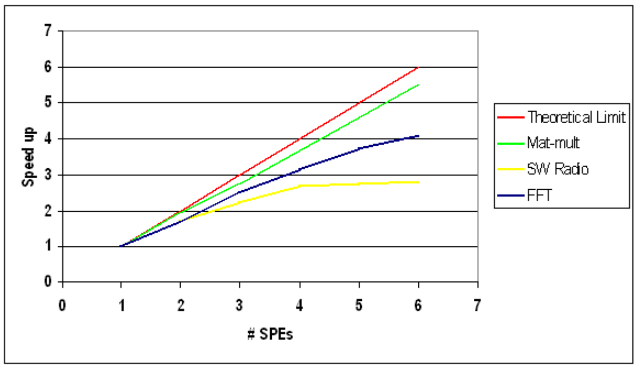


Figure 7.23: Benchmark results. Speedup is normalized against the execution with 1 SPE.

We carried on our analysis on a Sony PlayStation 3, which represents an inexpensive solution to work with Cell processor. The performance results for the three examples are shown in Figure.7.23. The figure presents the performance results obtained when running an increasing number of SPEs scaled to the case when 1 SPE is used. The Mat-mult benchmark scales almost perfectly w.r.t. the theoretical speed-up limit, thus proving the efficiency of our run-time environment and its almost negligible overhead. Also in the case of FFT an increasing number of SPEs brings to perceptible speed-ups. The software radio benchmark instead shows good speedup until only three SPEs: there is a path in the graph which duration bounds the speed up. More performance improvement can be reached in this case using software pipeline optimization.

7.7.3 Validation of optimizer solutions

To analyze the quality of our optimizer allocator, we performed experiments on a large set of synthetically generated task graphs. A task-graph generator has been implemented, so that it is possible to obtain a large number of pseudo-random test cases. To explore applications with different characteristics, the generator can be configured to produce task graphs with specific features, such as:

- Number of tasks;
- Average number of communication arcs between tasks;
- Average queue buffer size;
- Buffer and program data location;
- Average task execution time.

For test purposes, we produced three sets of task graphs: one with 15-task instances, one with 25-task instances and one with 30 tasks per instance. The test instances have then been processed by both our optimal and heuristic (list scheduling algorithm and Round-Robin allocator) solvers.

At this point, we had to profile the behaviour of applications (we were mainly interested in task execution times). Application profiling can be easily done running the applications on IBM Full-System Simulator, or using the profiling tool that comes with the Cell SDK, but the slowness of the former and the inaccuracy of the latter would prevent to run computationally intensive and precise tests. The best choice was to run the code on real hardware. Thus, all our experimental tests have been conducted using all the available SPEs (i.e. six for PlayStation 3): this is the worst case in terms of synchronization, communication and bus usage, as well as complexity of scheduling and allocation problem.

We compared for each instance the heuristic allocation and scheduling with the optimal ones. Figure.7.24 shows the percentage difference (normalized on

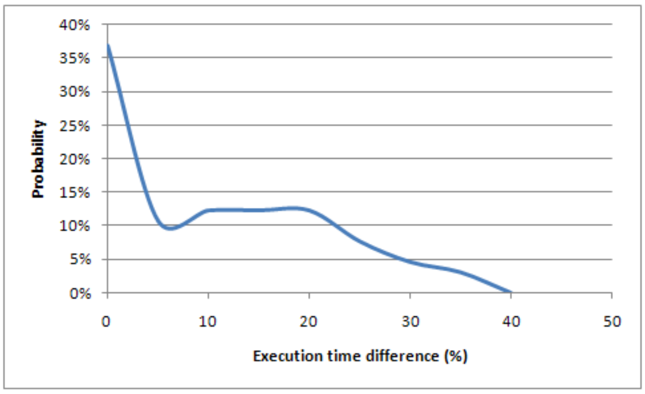


Figure 7.24: Histogram of the optimality gap on 100 instances.

100 instances) of heuristic solutions with respect to the optimal ones. For the 37% of the instances, the heuristic and the exact solutions matched, for the remaining instances the heuristic optimizer produced sub-optimal results, with up to a 35% optimality gap.

Figure.7.25 represents the average performance (application execution time) for each set of tests (15, 25 and 30 tasks). This proves that the error does not grow too rapidly with the number of tasks, but remains around 15%. These experiments confirm that the optimal solver achieves significant better results, but also that the list scheduler with round-robin allocator provide resonable solutions for critical instances.

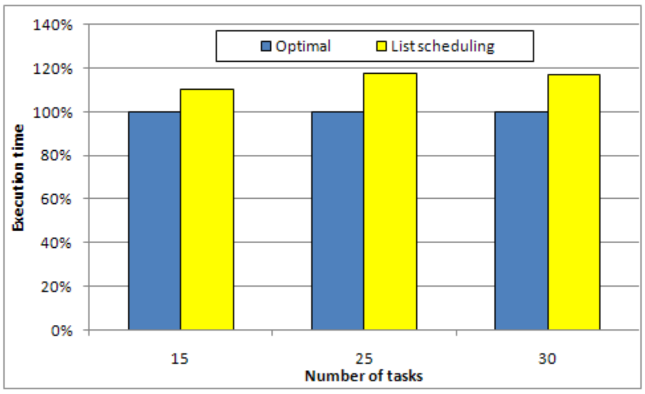


Figure 7.25: Comparison of average application execution times.

7.8 Conclusions

We propose a complete framework, called Cellflow, to help programmers in software implementation on the Cell Broadband Engine processor. Cellflow is composed by an off-line development framework and an on-line runtime support, and experimental results demonstrate the efficiency and viability of our solution.

Bibliography

[1] Benini, L., Bertozzi, D., Guerri, A., Milano, M.: Allocation and scheduling for MP-SOCs via decomposition and no-good generation. In: Proc. of the Int.l Conference in Principles and Practice of Constraint Programming (CP 2005).

[2] Benini, L., Bertozzi, D., Guerri, A., Milano, M.: Allocation, Scheduling and Voltage Scaling on Energy Aware MPSoCs. In: Proc. of the Int.l Conference on Integration of Artificial Intelligence and Operations Research techniques in Constraint Programming (CPAIOR 2006).

[3] L.Benini, M. Lombardi, M. Mantovani, M. Milano and M. Ruggiero Multi-stage Benders Decomposition for Optimizing Multicore Architectures. Technical Report LIA-008-07.

[4] Bockmayr, A., N. Pisaruk. Detecting infeasibility and generating cuts for MIP using CP. Int. Workshop Integration AI OR Techniques Constraint Programming Combin. Optim. Problems CP-AI-OR03, Montreal, Canada, 2003.

[5] B. Flachs and et al. A streaming processing unit for a cell processor. In: Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International, pages 134–135, 2005.

[6] I. E. Grossmann and V. Jain. Algorithms for hybrid milp/cp models for a class of optimization problems. INFORMS Journal on Computing, 13:258–276, 2001.

[7] Hooker, J.N., Ottosson, G.: Logic-based benders decomposition. Mathematical Programming 96 (2003) 33–60

[8] J. N. Hooker. A hybrid method for planning and scheduling. In: Proc. of the 10th Intern. Conference on Principles and Practice of Constraint Programming - CP 2004, pages 305–316, Toronto, Canada, Sept. 2004. Springer.

[9] J. N. Hooker. Planning and scheduling to minimize tardiness. In: Proc. of the 11th Intern. Conference on Principles and Practice of Constraint Programming - CP 2005, pages 314–327, Sites, Spain, Sept. 2005. Springer.

[10] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. IEEE Micro, 26(3):10–23, 2006.

[11] D. Pham and et al. The design and implementation of a first-generation cell processor. IEEE International Solid-State Circuits Conference ISSCC. 2005, pages 184–592 Vol. 1, 2005.

[12] R. Sadykov, L.A. Wolsey. Integer Programming and Constraint Programming in Solving a Multimachine Assignment Scheduling Problem with Deadlines and Release Dates. INFORMS Journal on Computing Vol. 18, No. 2, 2006, pp.209-217.

[13] J. L. de Siqueira N. and J.F. Puget. Explanation-Based Generalisation of Failures. European Conference on Artificial Intelligence, 1988, 339-344.

[14] U. Junker QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems In Proc. of the Nineteenth National Conference on Artificial Intelligence - AAAI 2004, pages 167–172, San Jose, California, USA, Jul. 2004, AAAI Press / The MIT Press

[15] Lombardi, M. and Milano, M. Stochastic Allocation and Scheduling for Conditional Task Graphs in MPSoCs In: Proc. of the Int.l Conference in Principles and Practice of Constraint Programming, 2006

[16] Tarim, A. and Miguel I. A Hybrid Benders' Decomposition Method for Solving Stochastic Constraint Programs with Linear Recourse In: Proc. of CSCLP 2005, pages 133–148, Springer

[17] Cambazard, H. and Jussien, N. Integrating Benders Decomposition Within Constraint Programming In: Proc. of the Int.l Conference in Principles and Practice of Constraint Programming, 2005, pages 752–756, Springer

[18] Ibm cell broadband engine software development kit, http://www.alphaworks.ibm.com/tech/cellsw/download.

[19] D. A. Bader, V. Agarwal, and K. Madduri. On the design and analysis of irregular algorithms on the cell processor: A case study of list ranking. In *IPDPS*, pages 1–10. IEEE, 2007.

[20] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. Cellss: a programming model for the cell be architecture. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 86, New York, NY, USA, 2006. ACM Press.

[21] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation. In *IBM White paper*, 2005.

[22] A. Eichenberger and et al. Optimizing compiler for the cell processor. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[23] A. E. Eichenberger and et al. Using advanced compiler technology to exploit the performance of the cell broadband enginetm architecture. *IBM Syst. J.*, 45(1):59–84, 2006.

[24] H. El-Rewini, H. H. Ali, and T. Lewis. Task scheduling in multiprocessing systems. *Computer*, 28(12):27–37, 1995.

[25] B. Flachs and et al. A streaming processing unit for a cell processor. In *Solid-State Circuits Conference, 2005. Digest of Technical Papers. ISSCC. 2005 IEEE International*, pages 134–135, 2005.

[26] H. Hofstee. Cell broadband engine architecture from 20,000 feet. In *IBM White paper*, 2005.

[27] X. R. A. Jimenez-Gonzalez, D.; Martorell. Performance analysis of cell broadband engine for high memory bandwidth applications. *Performance Analysis of Systems and Software, 2007. ISPASS 2007. IEEE International Symposium on*, pages 210–219, 25-27 April 2007.

[28] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *IEEE Micro*, 26(3):10–23, 2006.

[29] D. Kunzman, G. Zheng, E. Bohm, and L. V. Kale. Charm++, offload api, and the cell processor. In *Proceedings of PMUP Workshop at PACT'06*, 2006.

[30] L. kuo Liu, S. Kesavarapu, J. Connell, A. Jagmohan, L. Leem, B. Paulovicks, V. Sheinin, L. Tang, and H. Yeo. Video analysis and compression on the sti cell broadband engine processor. *Multimedia and Expo, 2006 IEEE International Conference on*, pages 29–32, 9-12 July 2006.

[31] S. Maeda, S. Asano, T. Shimada, K. Awazu, and H. Tago. A real-time software platform for the cell processor. *IEEE Micro*, 25(5):20–29, 2005.

[32] M. D. McCool. Data-parallel programming on the cell be and the gpu using the rapidmind development platform. In *GSPx Multicore Applications Conference*, 2006.

[33] M. Ohara, H. Yeo, F. Savino, G. Iyengar, L. Gong, H. Inoue, H. Komatsu, V. Sheinin, S. Daijavaa, and B. Erickson. Real-time mutual-information-based linear registration on the cell broadband engine processor. *Biomedical Imaging: From Nano to Macro, 2007. ISBI 2007. 4th IEEE International Symposium on*, pages 33–36, 12-15 April 2007.

[34] F. Petrini, G. Fossum, J. Fernndez, A. L. Varbanescu, M. Kistler, and M. Perrone. Multicore surprises: Lessons learned from optimizing sweep3d on the cell broadband engine. In *IPDPS*, pages 1–10. IEEE, 2007.

[35] D. Pham and et al. The design and implementation of a first-generation cell processor. *IEEE International Solid-State Circuits Conference ISSCC. 2005*, pages 184–592 Vol. 1, 2005.

[36] V. Sachdeva, M. Kistler, W. E. Speight, and T.-H. K. Tzeng. Exploring the viability of the cell broadband engine for bioinformatics applications. In *IPDPS*, pages 1–8. IEEE, 2007.

[37] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM Press.

[38] X. D. Zhang, Q. J. Li, R. Rabbah, and S. Amarasinghe. A lightweight streaming layer for multicore execution. In *Workshop on Design, Architecture and Simulation of Chip Multi-Processors*, 2007.

[39] U. Junker. QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained. In *Problems In Proc. of the Nineteenth National Conference on Artificial Intelligence – AAAI 2004*, pages 167–172, San Jose, California, USA, Jul. 2004, AAAI Press – The MIT Press.

Chapter 8

Portable Device Display Technologies

8.1 Overview

Display technologies are relatively new. The cathode ray tube was developed less than 100 years ago. For the last 10 years, scientists and engineers have been working closely to create a display technology capable of providing a paper and ink like reading experience, with superior viewability, but also with respect to cost, power, and ease of manufacture.

An LCD display system is composed of an LCD panel, a frame buffer memory, an LCD controller, and a backlight inverter and lamp or light-emitting diode (LED). High-resolution, high-color LCDs require large LCD panels, high-wattage backlight lamps, and large-capacity frame buffer memories, which together lead to high-power consumption.

The processor and the menory are in power-down mode during the slack time, but the display components are always active mode, for as long as the display is turned on. This makes the LCD backlight the dominant power consmer, with the LCD panel and the frame buffer coming a second and third in power consumption. A modern mobile device requires a lot of computing power. With interactive applications, such as a video telephony or an assisted GPS, an even higher portion of the energy will be consumed by the display system.

8.2 Mobile Device Power Distribution

Figures 8.1, 8.2, 8.3 indicate pie charts that illustrate the mobile device power distribution ranging from a legacy mobile device (voice only) to a smartphone/multimedia

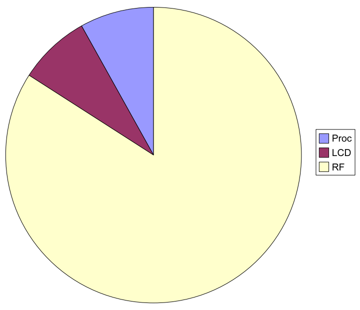


Figure 8.1: Legacy Handset Power Distribution Radio Frequency (RF) Dominated Power Consumption in Legacy (Voice Only) Handsets

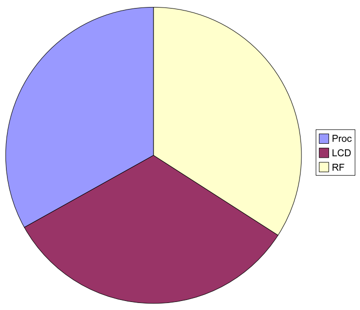


Figure 8.2: Feature Rich Handsets Power Distribution. More Equitable Power Consumption Distribution in Smartphone/Multimedia Mobile Devices

mobile device to a gaming targeted mobile device. Convergence of features is driving new application processing and visual display requirements.

8.3 Backlights

Backlght mechanical design has become very sophisticated, allowing very few LEDs to be used with highly complex optical light pipes/light spreads which

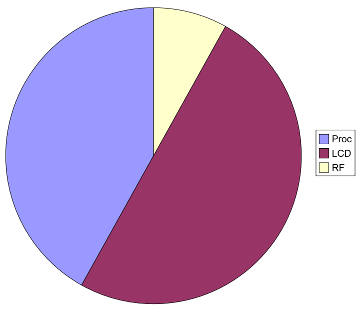
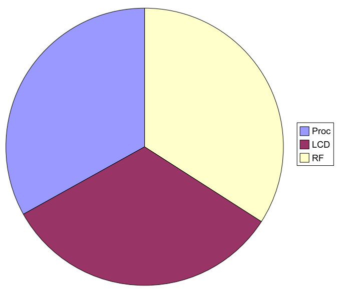
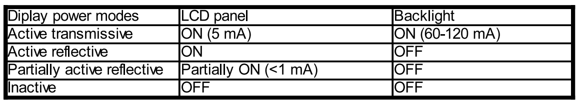


Figure 8.3: Game Oriented Phone Power Distribution. Power Distribution for Single Game Players, Dominated by the Display and Processing





Diplay power modes	LCD panel	Backlight
Active transmissive	ON (5 mA)	ON (60-120 mA)
Active reflective	ON	OFF
Partially active reflective	Partially ON (<1 mA)	OFF
Inactive	OFF	OFF

Figure 8.4: Most Power is Consumed When the Backlight is On

create uniform illumination of the LCD. Power consumption of the backlight is a critical issue. Now that white LEDs have replaced cold-cathode fluorescent lamps (CCFLs) for backlighting mobile device displays (and LEDs will soon replace CCFLs in laptop also), the next major innovation is the RGB LEDs, which will significantly enlarge the color gamut of the backlight, and therefore the display itself. It is claimed that an image that has higher color saturation (larger color gamut) looks brighter than a lower color saturation image. Engineers will take advantage of this effest to lower backlight power consumption. The RGB LED backlight will be a great benefit to mobile device displays.

Many mobiel devices are equipped with color thin-film transistor (TFT) liquid-crytal displays (LCDs). A quality LCD system is now the default configuration for handheld embedded system. An LCD panel does not illuminate itself and thus requires a light source. A transmissive LCD uses a backlight, which is on of the greadiest consumer of power of all system components. A reflective LCD uses ambient light and a reflector instead of the backlight. However, reflective LCD is not siutable for quality displays, and complementary use f ambient light and the backlight, named transflective LCD, is used for small handheld devices. When the backlight is turned off, a transmissive LCD displays nothing but black screen; even transflective screens are barely legible without the backlight.

Most useful applications require the backlight to be on. Figure 8.4 indicates the display power modes and the current drawn when the backlight is turned on [1].

Figure 8.5 indicates the system level components contribution to the power consumption. Note the power consumend by the backlight.

There are many techniques employed to conserve energy consumed by a display system. Ambient luminance affects the visibility of LCD TFT panels. However, by taking account of this, backlight autoregulation [2] can also reduce the average energy requirements of the backlight. Simultaneously brightness and contrast scaling [4] further enhances image fidelity with a dim back-

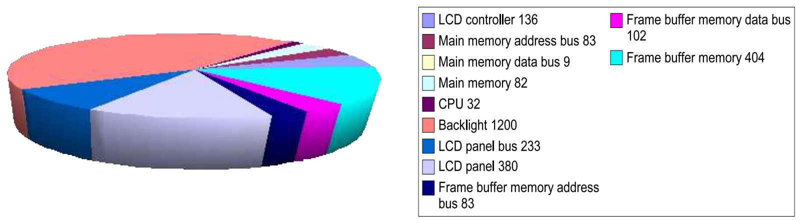


Figure 8.5: System Wide Power Consumption [1]

light, and thus, permits an additional reduction in abcklight power. Even more aggressive management of the backlight can be achieved by modification of the LCD panel to permit zoned backlighting [5]. Additional energy conserving techniques include dynamic luminance scaling (DLS), dynamic contrast enhancement (DCE), and backlight autoregulation. These techniques will be discussed later in this chapter.

8.4 Display Technologies

Display technologies such as backlight LCDs, reflective LCDs, electroluminescent (EL) displays, organic LEDs (OLEDs), and electrophoretic displays (EPD) objective is to achieve paper-like viewing displays.

There are four primary approaches to flat-panel displays. Three are illustrated in Figure 8.6:

1. Transmissive displays work by modulating a source of light, such as a backlight, using an optically active material such as a liquid-crystal mixture.

2. Emissive displays such as OLEDs make use of organic materials to generate light when exposed to a current source.

3. Reflective displays work by modulating ambient light entering the display and reflecting it off a mirror-like surface. Until recently, this modulation has typically been accomplished using liquid-crystal mixtures or electrophoretic mixture.

4. Transflective displays are a hybrid combination of a transmissive and reflective display. This technology was developed to rpovide sunlight

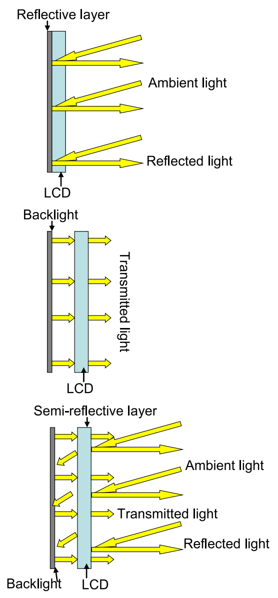


Figure 8.6: Approaches to Displays

Existing Technologies

Parameter	Units	Reflective STN	Reflective TFT	Backlight TFT	EPD	OLED
Color		B&W	Color	Color	Limited Color	Color
Resolution		Low	Good	Good	Good	Good
Sunlight readable	80% contrast	Yes,good	Yes,good	No	Yes,excellent	Yes,poor
Video response speed	ms	100	20	20	500	1/1,000
Static power including drivers	mW	15	20	200	0	200
Thickness	mm	5	6	9	1.25	3
Environmental	Temperature Range (°C)	10 - 30	10 - 30	10 - 30	0 - 50	10 - 50

Figure 8.7: Comparison of Display Technologies

viewability for transmissive displays. Being a compromise however, this type of display technology offers a compromised viewing experience.

Reflective displays were invented primarily to address the shortcomings of transmissive and emissive displas, namely power consumption and poor readability in bright environments.

Since transmissive LCDs require a power-hungry backlight and emissive OLEDs require a constant power source to generate light, it makes it difficult for designers of these technologies to reduce power consumption. This is especially important for battery-powered portable devices such as mobile phones, PDAs, digital music players, digital cameras, GPS units, and mobile gaming devices. With efficient use of ambient light, reflective displays eliminate the backlight unit and offer both significant power savings and a thinner display module (8.7).

Mobile device display technoogies can be separated into emissive and non emissive classes. This classification is expanded and illustrated in Figure 8.8.

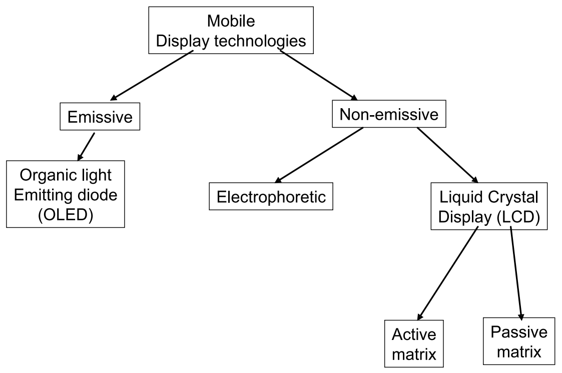

Mobile Display technologies
Emissive
Non-emissive
Organic light Emitting diode (OLED)
Electrophoretic
Liquid Crystal Display (LCD)
Active matrix
Passive matrix

Figure 8.8: Classification of Mobile Device Displays

8.5 TFT LCD

There are three types of TFT LCD panels. In transmissive LCDs, a backlight illuminates the pixels from behind. Transmissive LCDs offer a wide color range and high contrast, and are typically used in laptops. They perform best under lighting conditions ranging from complete darkness to an office environment.

Reflective LCDs are illuminated form the front. Reflective LCD pixels reflect incident light originating from the ambient environment or a frontlight. Reflective LCDs can offer very low-power cnsumption (expecially without frontlight) and are often used in small portable devices such as handheld games, PDAs, or instrumentation. They perform best in a typical office environment or in brighter lighting. Under a dim lighting, reflective LCDs require a frontlight.

Transflective LCDs are partially transmissive and partially reflective, so they can make use of environment light or backlight. Transflective LCDs are common in devices used under a wide variety of lighting conditions, from complete darkness to sunlight.

Transmissive and transflective LCD panels use very bright backlight sources that emit more than 1,000 cd/m2. However, the transmittance of the LCD is relative low, and thus the resultant maximum luminance of the panel is usually less than 10% of the backlight luminance.

Theoretically, the backlight and the ambient light are additive. However, once the backlight is turned on, a transflective LCD panel effectively operates in the transmissive mode because the backlight source is generally much brighter than the ambient light.

As stated earlier, backlighting for LCDs is the single biggest power draw in portable displays. This is especially true in bright environments where the backlight has to be switched to the brightest mode. Given how difficult it is to view a typical transmissive LCD in a sunlit environment, LCD developers

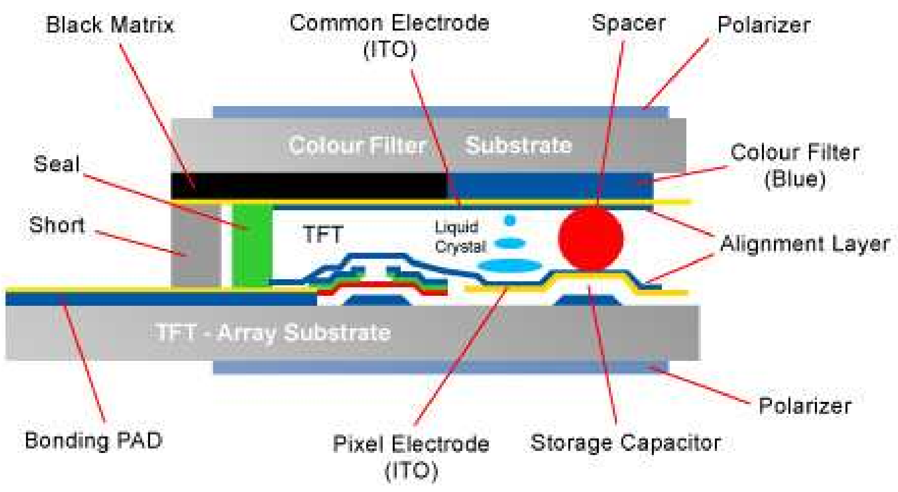


Figure 8.9: LCD Structure

have been working diligently on reflective LCDs.

Currently there are a number of portable devices using transflective LCDs. The transflective display was invented to improve the performance of the transmissive LCD outdoors, where bright ambient light quickly overpowered the LCD backlight, making the display hard to read. It was also configured to address the shortcomings of a purely reflective LCD in a dark environment. The transflective display employs a reflector that lets some light through from a backlight. Using such an element, the display can be used in the dark where the backlight provides illumination through the partly transmissive reflecting element. In the bright outdoors, the backlight can be switched off to conserve power and the mirrored portion of the reflector allows the LCD to be viewed by making use of the ambient light. Theoretically, the transflective display appears to fix the shortcomings of the purely reflective and transmissive displays. But in reality, this approach is a compromise and offers poor viewing experience.

Figure 8.9 shows the complexity of an LCD. The extensive use of optical films such as polarized and color filters, as well as the TFT element which itself requires several process step fabricate. Since LCDs work with polarized light, the necessity of using a polarizer limits the amount of light that is reflected or transmitted from the display. The additional layers, such as the color filter, reduce light even further. Consequently, today's LCDs require brighter backlight in order to be readable, whether in total darkness or in the bright sunlight. These brighter backlights lead to greater power consumption.

Despite the ever increasing advantages in LCD's technology, their power

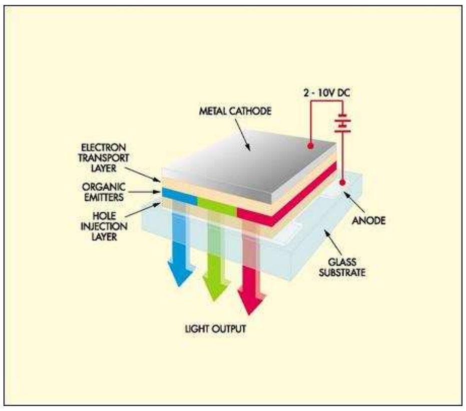


Figure 8.10: OLED Structure

consumption is still one of the major limitations to mobile. There is a clear trend towards the increase of LCD size to exploit the multimedia capabilities of mobile devices that can receive and visualize high deifnition video and pictures. Multimedia applications running on these deivces impose LCD screen sizes of 2.2-3.5 in. and more to display video sequences and pictures with the required quality.

8.6 OLED

Similar to LCDs, OLEDs can be constructed using a passive or active matrix. The basic OLED cell structure is comprised of a stack of thin organic layers that are sandwiched between a transparent anode and a metallic cathode. When a current passes between the cathode and anode, the organic compounds emit light (see Figure 8.10). Unlike LCDs, passice matrix OLEDs does not suffer from lower contrast or slower response time. However, OLEDs offer several advantages over LCDs.

The obvious advantage is that OLEDs are like tiny light bulbs, so they do not need a backlight or any other external light source. They are less than one-third of the bulk of a typical color LCD and about half the thickness of most black-and-white LCDs. The viewing angle is also wider, about 160. OLEDs also switch faster than LCD elements, producing a smoother animation. Once initial investments in new facilities are recouped, OLEDs can potentially compete at an equal or lower cost than incumbent LCDs.

Despite these advantages, OLEDs have a relatively short lifespan and as power/brightness is increased the life is reduces dramatically. This is especially true for the blues, which lose their color balance over time. In addition, only low-resolution OLED displays can use passive matrix backplanes and higher resolutions require active matrices, which need to be highly conductive since OLEDs are current driven. Typically, low temperature poly silicon (LTPS) backplanes are used which adds cost and complexity. These conductors are also highly reflective requiring the OLED designers to add a circular polarizer on the front if the display reducing the efficiency if the display and increasing the cost. Finally, as is the case with all emissive displays, OLED displays have poor readability in environments such as the bright outdoors.

Bibliography

[1] T.Botzas. PenTile RGBW display technology for meeting aggressive power budgets in high resolution multimedia mobile applications. In International Wireless Industry Consortium. 2005.

[2] I.Choi. J2ME LBPB (Low Power Basis Profile of the Java 2 Micro Edition) Computer Systems Laboratory, 2002

[3] F.Gatti and A.Acquaviva and L.Benini and B.Ricco. Low power control techniques for TFT LCD displays. In Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, 2002

[4] W.C.Cheng and Y.Hou and M.Pedram. Power minimization in a backlit TFT-LCD display by concurrent brightness and contrast scaling In Proceedings of DATE 04, 2004

[5] J.Flinn and M.Satyanarayanan. Energy-aware adaptation for mobile applications. In Proceedings of the Symposium on Operating Systems Principles, 1999.

Chapter 9

Low Power LCD Techniques

9.1 Overview

A number of low-power LCD techniques have been investigated. These include DLS, extended DLS (EDLS), frame buffer compression, dynamic color depth control, variable duty ratio refresh, abcklight autoregulation, and dark window optimization. Each techniques saves the power consumption of the display system by reducing the activity of the corrensponding components such as the backlight luminance, the color depth, the refresh duty ratio, and the pixel brightness.

9.2 Dynamic Luminance Scaling

DLS keeps the perceived intensity or contrast of the image as close as possible to the original while achieving significant power reduction. DLS compromises quality of image between power consumption, which fulfills a large variety of user preferences in power-aware multimedia applications. DLS saves 20-80% of power consumption of the backlight system while keeping a reasomable amount of image quality degradation.

DLS adaptively dims the backlight with appropriate image compensation so that the user perceives similar levels of brightness and contrast with minor image distortion.

The luminance of the backlight is proportional to its power consumption. As we dim the backlight, the brightness of the image on the LCD panel is reduced, but we save power. The principle of DLS is to save power by backlight dimming while restoring the brightness of the image by appropriate image compensation [3], [4].

Brute-force backlight dimming is a traditional technique to save power consumption, but it reduces brightness, and thus, the display quality is degraded. By contrast, DLS does not sacrifice the overall brightness of the image but accomodates minor color distortions. To achieve the maximum power saving for a given color distortion limit, DLS dynamically scales the luminance of the backlight as the image on the LCD panel changes.

There are a number of image compensation algorithms described in the following section.

9.3 Brightness Compensation

Brightness is the intensity of an image perceived by human eyes. As an approximation brightness is considered linearly proportional to the luminance of the LCD panel. The major computational overhead of brightness compensation is the construction of the transformation function and the transformation of each pixel color. Building the transformation function includes the construction of the histogram and determining a value for the threshold. Transforamtions are typically performed either by moltiplication and division operations.

Brightness compensation allows a significant degree of backlight dimming while keeping the distortion ratio reasonable, as long as the image has a continous histogram which is not severely skewed to bright areas. Although all the histograms are dicrete by definition, we express that a histogram is continuos if adjacent value are similar with each other, considering the original image before digitization of the color values. Discrete histograms generally make it difficult to determine a proper threshold value and most graphical user interface (GUI) components have dicrete histograms.

9.4 Image Enhancement

Image enhancement allows one to apply DLS for the images with discrete histograms where the brightest area dominates the image. Techniques of histogram stretching and histogram equalization are employed. Histogram stretching is an extension of brightness compensation, in that the histogram is stretched with respect to the low threshold as weel as the high threshold. Histogram stretching truncates data not only in the brightest areas but also in the darkest areas. It generally doubles the amount of backlight dimming that can be achieved, in comparison with brightness compensation. Histogram stretching implies contrast enhancement rather than recovery of brightness. The contrast enhancement is more desirable for GUI applications, where readability is the primary objective.

Histogram stretching outperforms brightness compensation. However, building the transformation function has twice the computational complexity. Sometimes, the colors of objects are not important, but we need maximum readability. Text-based screens often fall in this category. In such cases, we need to achieve more contrast to allow more backlight dimming. Histogram equalization is a usefull technique for this porpuse.

Image enhancement is also applicable to image with continous histograms. Since histogram stretching is an extension of brightness compensation, there is similar distortion to the image. The majority of pixels preserve their original colors, and thus we can apply brightness compensation and histogram stretching for streaming images where inter-frame consistency must be considered. On the other hand, histogram equalization is not applicable to streaming images because in this case most pixels change their colors. Histogram equalization has a tendency to spread the histogram of the original image so that the levels of the histogram-equalized image span a wider range [5].

Histogram equalization generally offers better readability than histogram stretching when the image has a discrete spectrum. The computational complexity for building the transformation function of histogram equalization is the same as that for brightness compensation. Since the transformation funciton is not a polynomial implementation, a table lookup is desiderable.

9.5 Context Processing

Histogram equalization generally outperforms histogram stretching in terms of readability for GUI applications, if the histogram is dicrete. However, some minor color may be merged into each other and are thus no longer distinguishable after histogram equalization. In the case of photographs, some minor colors may be merged into others ir become similar to each other. But, in the case of text the number of pixels does not correlate with importance. So we never allow text to be merged into its background after histogram equalization.

Context processing is a usefull technique to prevent small foreground objects from having similar color to their background after histogram equalization. If a foreground color and a background color become equal or similar after histogram equalization, context processing re-stretches their colors so that the distance between them in color space is a maximum.

Context processing is a post-processing step that can be applied after brightness compensation, histogram stretching, or histogram equalization. Distortion ratio no longer has meaning if context processing is used. Context processing does not require the overhead of building a transformation function since it is not based on the histogram. However, transformation does require con-

Panel Mode	Transmissive mode		Reflective mode
Backlight	Full	Dimmed	No backlight
Image	Original image	Brightness enhancement	Contrast enhancement
Power source	External power supply	Moderate battery power	Poor battery power

Figure 9.1: EDLS Framework [1]

text information for the application. Context processing can be implemented by addition operation only.

DLS consists of two major operation: image compensation and backlight control. Both can be implemented by modifying the application program and operating system or the frame buffer device driver.

9.6 Extended DLS

DLS is extended to cope with transflective LCD panels, which operate both with and without a backlight, depending on the remaining battery energy and ambient luminance. These popular transflective LCD panels are the dominant choice for battery operated electronic systems because they allow an image to remain visible without a backlight, even though the quality can be poor.

Remember the principle of DLS is to reduce the light source's luminance but compensate for the loss in brightness by allowing more light to pass through the screen, enhancing the image luminance. The viewer should perceive little change. DCE also enhances image quality under a dimmed backlight, but does so by increasing the image's contrast. DCE requires similar image processing to DLS, and thus we have the same degree of freedom in adaptation. Although DLS preserves the original colors, DCE results in a noticeable change to the original colors in pursuit of higher contrast and improved legibility. DCE is a very aggressive power management scheme for transmissive LCD panels, which differentiates it from DLS. The EDLS framework, as illustrated in Figure 9.1, achieves a congruent combination of DLS and DCE.

Fundamentally the EDLS interface is a simple slider knob. The EDLS knob controls the trade-off between energy consumption and image quality. In addition it provides users with a power management scheme that can extend battery life at the cost of whatever Quality-of-Service (QoS) degradation the user will accept. There is also automatic mode that changes the power management setting, depending on the remaining battery energy.

When connected to an external power source, the backlight is fully on and exhibits its maximum luminance. There should be no backlight power management so that user can enjoy the best image quality. When the system is battery powered, however, user might want to extend the battery life for future use, even if the battery is already fully charged. But users generally are not ready to sacrifice appreciable picture quality at that stage. As the remaining battery energy decreases, users might become increasingly willing to compromise image quality to extend battery life. This is the point at which EDLS applies DLS.

With a poor power budget, the user's prime concern might well be to complete the current task within the remaining battery energy budget, even if the image quality decreases. This is the optimum time for EDLS to change from DLS to DCE mode. Although DCE might alter the original colors, a moderate degree of DCE does at least maintain a fixed distortion ratio. However, if the battery energy is nearly exhausted, the only remaining option is to turn off the backlight. Without the backlight, EDLS applies DCE to achieve the maximum possible contrast. In this case, EDLS cannot guarantee a fixed amount of image distortion, but the user should still be able to read the display and finish the task.

The EDLS process starts by building a red-green-blue histogram of the image for display. The EDLS slider determines the panel mode (transmissive or reflective), the image processing algorithm (DLS or DCE), and the maximum allowed percentage of saturated pixels, SR, after image processing. EDLSprocess derives upper and lower thresholds TH and TL from SR and the histogram, and calculates a scaling factor that controls the amount of backlight dimming.

EDLS significantly reduces backlight power consumption. However, it results in power, delay, and area overhead that take place in other components. These overheads are primarily determined by the screen resolution, refresh rate, and color depth.

9.7 Backlight Autoregulation

A mobile device operating in an environment with low ambient luminance, and this luminance can be detected by a photo sensor, the backlight can be dimmed without effecting the user. Backlight autoregulation adaptively dims the backlight in response to changes in the ambient luminance [2]. Backlight autoregulation is applicable while maintaning QoS only when reduced contrast by backlight dimming does not compromise the visibility. The contrast between the LCD panel with normal backlight and the dark environment with low ambient luminance is high enough so that we can safely reduce the contrast by dimming the backlight without compromise the visibility. To take advantage of backlight autoregulation, a mobile device must be equipped with a photo sensor to detect the ambient luminance. Tpically the photo sensor can be the on board camera.

9.8 Frame Buffer Compression

Frame buffer compression [6] is used to reduce the power consumption of a frame buffer memory and its associated buses. LCD controllers periodically refresh theis display at 60 Hz activating the frame buffer.

Frame buffer compression reduces the activity of the frame buffer and thus its power consumption. The compression algorithm employed is based on run-length encoding for on-the-fly lossless compression.

An adaptive and incremental re-compression scheme, to accomodate frequent partial frame buffer updates efficiently, has been developed. The result is a savings from 30% to 90% frame buffer activity on average for various mobile applications. The implementation of compression scheme consumes 30 mW more power and 10% more silicon space than a conventional LCD controller without frame buffer compression. Howeverm, the power saved in the frame buffer memory is up to 400 mW.

9.9 Dynamic Color Depth

Dynamic color depth control [3] modifies the pixel organization in the frame buffer, wihch enables haf of the frame buffer memory devices to go into power-down mode at the cost of a decreased color depth.

Dynamic color depth control achieves an energy saving from the frame buffer. Variable duty ratio refresh [3] reduces the duty ratio of refresh cycles as far as possible. This occurs only if the time constraint of the storage capacitor of a sub-pixel on the TFT LCD panel is higher than the refresh period, saving power in the frame buffer and the LCD panel interface bus.

Engineers have been working onbacklight autoregulation [2], which adaptively dims th ebacklight in response to changes in the ambient luminance, and a dark window optimization [7] which modifies the windowing environment to allow changes to the brightness and color of areas of the screen that are not of current interest to the user. This saves power in OLED display panels.

There are many techniques available for low-power display systems. Choosing the proper techniques are very important. Frame buffer compression and dynamic color depth control have the same goal of reducing the power consumption from the frame buffer. However, they cannot both be applied at the same time, and so we have to select one. The user may be willing to allow some decrease un color depth in exchange for higher contrast in a document viwer, where image legibility is the most important QoS requirement, and dynamic color depth control can meet user's preferences. But if a photo image viewer is running, then image fidelity should e preserved, and we should adopt frame buffer compression.

Bibliography

[1] I.Choi. J2ME LBPB (Low Power Basis Profile of the Java 2 Micro Edition) Computer Systems Laboratory, 2002

[2] F.Gatti and A.Acquaviva and L.Benini and B.Ricco. Low power control techniques for TFT LCD displays. In Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems, 2002

[3] I.Choi and H.Shim and N.Chang Low-power color TFT LCD display for handheld embedded systems In Preceedings of the International Symposium on Low-Power Electronics and Design, 2002

[4] I.Choi and H.Shim and N.Chang LBPB (Low Power Basis Profile of the Java 2 Micro Edition) Computer Systems Laboratory, 2002

[5] R.C.Gonzales and R.E.Woods Digital Image Processing ed. Reading, MA: Addison-Wesley, 1992

[6] H.Shim and N.Chang and M.Pedram A compressed frame buffer to reduce display power consumption in mobile systems In Proceedings of ASPDAC, 2004

[7] S.Iyer and R.Mayo and P.Ranganathan Energy-adaptive display system designs for future mobile environments In USENIX Association Proceedings of MobiSys 2003

Chapter 10

XEC-DLS

10.1 Introduction

Despite the ever increasing advances in Liquid Crystal Display's (LCD) technology, their power consumption is still one of the major limitations to the battery life of mobile appliances such as smart phones, portable media players, gaming and navigation devices. There is a clear trend towards the increase of LCD size to exploit the multimedia capabilities of portable devices that can receive and render high definition video and pictures. Multimedia applications running on these devices require LCD screen sizes of 2.2 to 3.5 inches and more to display video sequences and pictures with the required quality.

LCD power consumption is dependent on the backlight and pixel matrix driving circuits and is typically proportional to the panel area. As a result, the contribution is also likely to be considerable in future mobile appliances. To address this issue, companies are proposing low power technologies suitable for mobile applications supporting low power states and image control techniques.

Modern displays support multiple low power configurations corresponding to different functionalities, aiming to reduce the power contribution of the display circuitry. For example, a standby state can be defined where the internal power supply of the LCD panel is switched off but the external power is supplied to ensure a fast display turn-on. Moreover, part of the input signal conditioning logic can be switched off if an internal memory is used to display data on the screen when RGB input is not sent.

Very recently, a new image processing technology has been announced by Hitachi Semiconductor called RCCS (RGB Colour Control System) based on backlight control. It is aimed at decreasing the luminance of the backlight when darker images are displayed. To compensate for the luminance reduction, the corresponding level of signals input to the LCD is increased. Since consumption of LCD is mainly given by the backlight, this technique can be quite promising concerning achievable power savings. However, its impact on image quality has not been tested since the final product has not yet been delivered.

On the research side, several power saving schemes and algorithms can be found in literature. Some of them exploit software-only techniques to change the image content to reduce the power associated with the crystal polarization, some others are aimed at decreasing the backlight level while compensating the luminance reduction by compensating the user perceived quality degradation using pixel-by-pixel image processing algorithms. The major limitation of these techniques is that they rely on the CPU to perform pixel-based manipulations and their impact on CPU utilization and power consumption has not been assessed.

We present an alternative approach that exploits in a smart and efficient way the hardware image processing unit (IPU) integrated in Freescale's multimedia application processors to implement a hardware assisted image compensation that allows dynamic scaling of the backlight with a negligible impact on QoS. The proposed approach overcomes CPU-intensive techniques by saving system power without requiring either a dedicated display technology or hardware modification. CPU processing, based on frame by frame histogram analysis on YUV image format, is minimized by means of hardware assisted downsizing and image processing tasks.

We provide a real implementation of the dynamic backlight scaling technique, embedded within the Video4Linux software subsystem running on a Freescale prototype development board (Advanced Development System) based on the i.MX31 application processor and a 3.3-inch QVGA display. By instrumenting this platform, we carried out a full characterization of both LCD and CPU power consumption. To properly assess the effectiveness of the proposed technique, a video player application and a variety of video sequences were used in a comparative test against a software only solution. Results show power savings up to 50% considering both LCD and CPU contribution with bounded QoS degradation and real-time performance guarantees.

The development of this technique can be described as follows: First, we demonstrate that energy efficient dynamic backlight scaling (DBS) is possible and is effective in reducing total power consumption of a real-life mobile platform. Second, we present a new backlight scaling technique overcoming the limits of state of the art research solutions based on CPU-intensive processing. Moreover, our work opens the opportunity for the comparison between a DBS solution based on commercial-off-the-shelf hardware with alternative

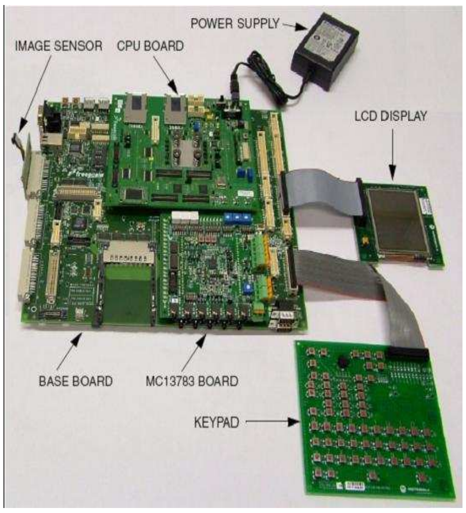


Figure 10.1: Freescale i.MX31 Advanced Development System (ADS)

solutions integrated in commercial LCD products. Finally, we show how DBS can be successfully integrated with an ambient light sensor to further improve power savings by adapting to environmental luminance conditions.

10.2 i.MX31 Application Development System

The i.MX31ADS enables the development of multimedia communication applications using the Freescale i.MX31, an ARM-11 based Application Processor, and the Freescale MC13783 Atlas audio and power management chip.

The ADS consists (Figure.10.1) of a base board with display and interface connectors, a CPU board with i.MX31 ARM-11 MCU, and a power management board with MC13783 Atlas chip. The system supports application software development, target board debugging and multiple interfaces for the addition of optional circuit cards. An LCD display panel, an image sensor, and a separate keypad are supplied with the ADS.

10.3 Multimedia Support in the Freescale i.MX31 Applications Processor

In the i.MX31 Applications Processor, the most computationally-intensive parts of video processing are offloaded from the ARM CPU and accelerated in hardware via Freescale's Image Processing Unit (IPU). This keeps power demands very low, but retains design flexibility.

The Image Processing Unit is designed to support video and graphics pro-

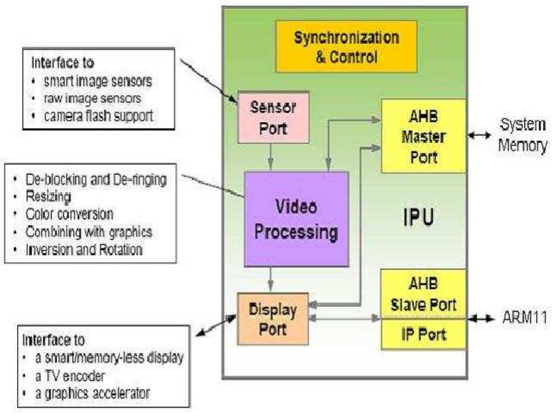


Figure 10.2: Message-oriented distributed memory architecture.

cessing functions and to interface to video/still image sensors and displays. A schematic block diagram of the IPU is presented in Figure.10.2.

The IPU includes all the functionality required for image processing and display management. This integrated approach yields several significant advantages.

The IPU is equipped with powerful control and synchronization capabilities to perform its tasks with minimal involvement of the ARM CPU. These include the following devices and capabilities:

- An integrated DMA controller (with two AHB master ports), allowing autonomous access to system memory;
- An integrated display controller, performing screen refresh of a memory-less display;
- A page-flip double buffering mechanism, synchronizing read and write accesses to the system memory to avoid tearing;
- Internal synchronization.

As a result, in most cases, the CPU involvement is limited to processing tasks such as video decoding, representing a significantly reduced processing load. In particular, for some situations which extend for long periods, such as screen refresh/update and camera preview, the ARM complex is idle and can be powered down, reducing considerably the power consumption and hence extending the battery life.

Moreover, the system-on-chip integration combined with internal synchronization, avoids unnecessary access to system memory, reducing the load on the memory bus and reducing further the power consumption. In particular,

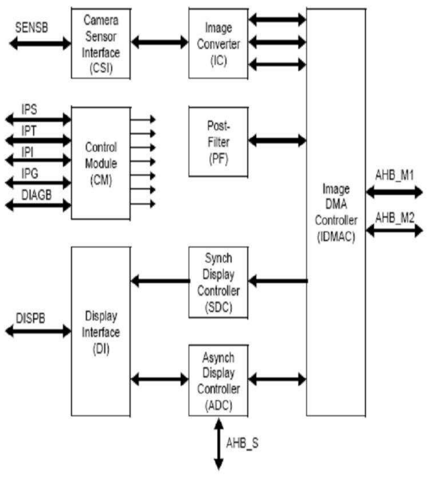


Figure 10.3: Image Processing Unit (IPU) block diagram.

output to a smart display can be processed on-the-fly, while reading from system memory.

Last but not least, the IPU performs some very processing-intensive image manipulations, adding considerable processing power to the system. Figure. 10.3 is a block diagram of the IPU.

The IPU consists of the CMOS Sensor Interface (CSI), the Image Converter (IC), the Post-Filter (PF), the Synchronous Display Controller (SDC), the Asynchronous Display Controller (ADC), the Display Interface (DI) and the Image DMA Controller (IDMAC). The sensor data is fed to the CSI. The IC executes pre- and postprocessing tasks. Pre- and postprocessing include the following operations:

- downsizing with independent integer horizontal and vertical ratios;
- resizing with independent fractional horizontal and vertical ratios;
- color space conversion (YUV to RGB, RGB to YUV, YUV to different YUV);
- combining a video plane with a graphics plane (blending of graphics on top of video plane);
- 90 degree rotation, up/down and left/right flipping of the image.

The PF implements the MPEG-4 and H.264 post-filtering algorithms (deblocking and deringing algorithms). The SDC is designed to support memoryless synchronous displays and synchronous interfaces of smart displays. The SDC combines video and graphics planes before sending data to a display.

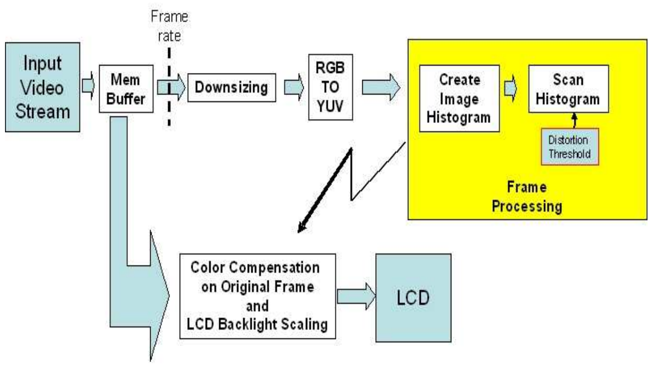


Figure 10.4: Dynamic Backlight Autoregulation block diagram.

Combining is performed with alpha-blending. The ADC is designed to support asynchronous displays.

10.4 Dynamic Backlight Luminance Scaling Technique

Dynamic backlight luminance scaling adaptively dims the backlight with appropriate image compensation so that the user perceives similar levels of brightness and contrast with minor image distortion. The luminance of the backlight is proportional to its power consumption. As we dim the backlight, the brightness of the image on the LCD panel is reduced, but we save power. The principle of dynamic backlight scaling is to save power by backlight dimming while restoring the brightness of the image by appropriate image compensation.

The dynamic backlight scaling scheme proposed does not sacrifice the overall brightness of the image but accommodates minor color distortions. To achieve the maximum power saving for a given color distortion limit, dynamic backlight scaling dynamically changes the luminance of the backlight as the image on the LCD panel changes.

The block diagram in Figure.10.4 describes how the framework operates. We can divide the overall flow into three main sections: preprocessing, processing and postprocessing.

During the preprocessing step, the input frame, a frame of a video stream or a simple still image, is modified in preparation for analysis by the CPU during the following processing step (as we will see in the analysis of section 5). We scale the size of the image and convert its input format to YUV format. These phases allow us to process a smaller image and thus to compute over a smaller set of data, as the YUV format already contains the luminance information of each pixel of the image.

After the preprocessing phase, the downsized and YUV converted image is fed to the processing step. In this section of the framework, each pixel is used to compute a luminance histogram of the preprocessed image. The luminance histogram is then analyzed to ascertain how much the image luminance may be increased, whilst satisfying a given distortion. The distortion is defined as the total number of saturated pixels after image compensation. The result of the processing step is to determine the amount of brightness reduction achievable on the backlight, that compensates the brightness increase that can be applied to the image itself in order to re-establish the correct perceived luminance level.

The final step of our framework is the postprocessing phase. It takes as input the original input frame, which is the same starting input image of the preprocessing phase with its original resolution and format. To this we apply the color compensation, modifying pixel per pixel the luminance of the overall frame. By applying the compensation to the original frame instead of the preprocessed one, we do not loose image quality, in terms of resolution and color distortion, due to resizing and format conversion.

10.5 Main Framework Settings

In our framework we can set and modify three key parameters, namely:

- frame rate;
- downsize ratio;
- distortion.

Tuning them we can find the optimal trade off between quality of service (QoS) and power savings.

The frame rate setting establishes how many frames per second will be fed to the main algorithm to calculate the new backlight and color compensation level. With a low frame rate we will use less power on CPU side in the frame analysis, since it will work less frequently, at the cost of a less responsiveness to average luminance changes of the input video stream to the framework.

The downsize ratio setting establishes how much we will downsize the input image during the preprocessing phase. A low downsize ratio will result in a low CPU utilization at the cost of a less precise luminance value calculation for color compensation.

The distortion setting establishes how many saturated pixels are admitted. A low distortion level means a good QoS in terms of final displayed image, but at the cost of a less aggressive overall power saving.

$$Z0 = 2^{SCALE-1} \cdot (X0 \cdot C00 + X1 \cdot C01 + X3 \cdot C02 + A0)$$
$$Z1 = 2^{SCALE-1} \cdot (X0 \cdot C10 + X1 \cdot C11 + X3 \cdot C12 + A1)$$
$$Z2 = 2^{SCALE-1} \cdot (X0 \cdot C20 + X1 \cdot C21 + X3 \cdot C22 + A2)$$

$$CSC1_{default} = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

10.6 IPU Enhanced Dynamic Backlight Auto-Compensation

Since several sections of our framework are computationally intensive, we take advantage of resources available on the iMX31 multimedia processor, thereby executing them in hardware. We use IPU features in the preprocessing and in the postprocessing phases. In the preprocessing step the IC submodule of the IPU is used to downsize the input frame and to convert its format to YUV, while in the postprocessing step we use the facilities of the SDC and PP submodules of the IPU to manipulate each pixel of the image, increasing its luminance, and thus to control the backlight level of the LCD display. We paid great attention in optimizing all memory accesses and CPU utilization. The communication between the CPU, main memory and IPU's submodules is made through DMA transfers, while synchronization is handled by interrupts.

10.6.1 Color compensation scaling the Color Space Conversion (CSC) Matrix

One of the main functions performed by the IPU is color space conversion which is done through the conversion matrix CSC1. The conversion matrix coefficients are programmable and they are stored in the IPU Task Parameter Memory.

The conversion equations are:

where X0, X1 and X2 are the component of the input format; Z0, Z1, Z2 the component of the output format; C00,..,C22 and A0,..,A2 the conversion matrix coefficients. All the parameters of the conversion matrix are written by the MCU to the Task Parameter Memory.

We decided to execute the color compensation in the postprocessing step scaling the values of conversion matrix coefficients.

For example, if input image format is RGB, the default conversion matrix is the identity matrix:

In order to compensate an image after backlight scaling down, we need to scale up the value of CS1default:

$$CSC1_{DLS} = \begin{matrix} b/b' & 0 & 0 \\ 0 & b/b' & 0 \\ 0 & 0 & b/b' \end{matrix}$$

Where b/b′ is the ratio between original backlight value (b) and scaled one (b′).

10.6.2 Ambient-aware Backlight Autoregulation

Optimal visibility requires different LCD backlight settings depending on the environmental light. Exploiting this consideration we implemented a mechanism that monitors ambient light and calculates an adequate value for the backlight level to be set. According to this policy the maximum backlight level will only be set in strongly lit environments (i.e. sunny outdoors), which is often not a likely operating condition. Only in this particular case we will still have the maximum power consumption. Any different environmental light condition will lead to heavy power savings, since, for the LCD panel under test, the LCD power consumption reduces linearly from over 600mW to less than 100 mW in low light conditions. We used two types of sensors. The camera-based one is used for compatibility reasons. The photo-diode is a commercial solution already.

This mechanism is implemented as a daemon inside a standalone kernel module. A kernel thread runs repeatedly once or more times per second and gathers information from the camera or photo-diode about the captured image luminance content. This allows us to derive an estimation of the environmental light. Based upon this information a separate algorithm computes an appropriate value for the LCD backlight.

10.6.3 Integration: Ambient-aware Dynamic Backlight Autoregulation

In this section we briefly describe how we integrated the Ambient-aware Backlight Autoregulation and the Dynamic Luminance Scaling schemes. The two techniques are completely independent but at the same time they interact with each other. The Dynamic Luminance Scaling scheme uses as reference point the backlight level set by the Ambient-aware Backlight Autoregulation. Once the Dynamic Backlight Luminance Scaling algorithm has computed the dimming gap for the backlight, it is applied starting from the optimal luminance value set by the Ambient-aware Backlight Autoregulation.

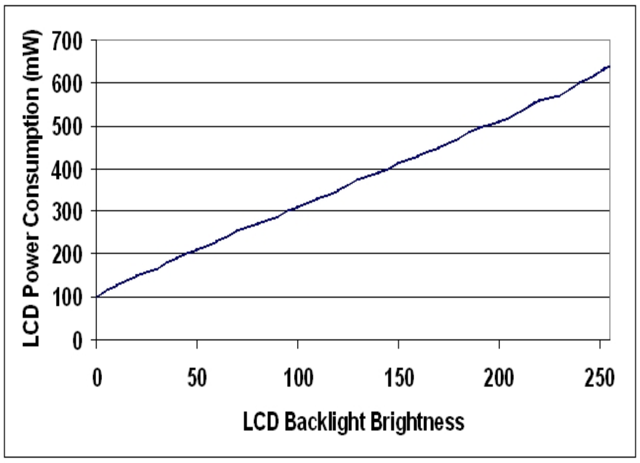


Figure 10.5: LCD consumption profiling varying the backlight luminance value.

10.7 Power and Performance Characterization

In order to understand if our proposed framework is sustainable, we carried out a feasibility study. We characterized the power consumption of the LCD panel and of the CPU during the processing of a frame.

The display we used was a 3,5-inch Sharp TFT LCD Module. Figure.10.5 shows the measurements results in terms of current and power dissipation of the LCD varying its brightness luminance.

LCD Power Consumption scales linearly with brightness scaling. A good power saving is potentially achievable, as the gap between the maximum and the minimum values is of 550 mW.

To understand the trade-offs between the achievable power savings on the LCD side versus the power wasted on the CPU, we analyzed the power consumption of the CPU during the processing of a frame. In this test the CPU had to scan each pixel in the frame in order to collect its luminance value.

Results are shown in Figure 10.6. As expected, the frame processing CPU utilization varies depending both on the frame resolution and the frame rate of the input video stream. So, with a frame rate of 30 fps it goes from 1% with a frame resolution of 80x64, to 12%, with a frame resolution of 320x240. While if we consider a frame rate of 1 fps, it goes from 0.2% to 1.2%. This analysis is very important both in terms of realtime performance and power consumption. Embedded multimedia devices are often required to respect realtime constraints and a backlight scaling technique not aware of this issue may lead to a violation of system requirements. Moreover, considering that the CPU consumes on average 350 mW during processing, while its consumption is almost negligible when idle (i.e. 10 mW), its power consumption while running is comparable

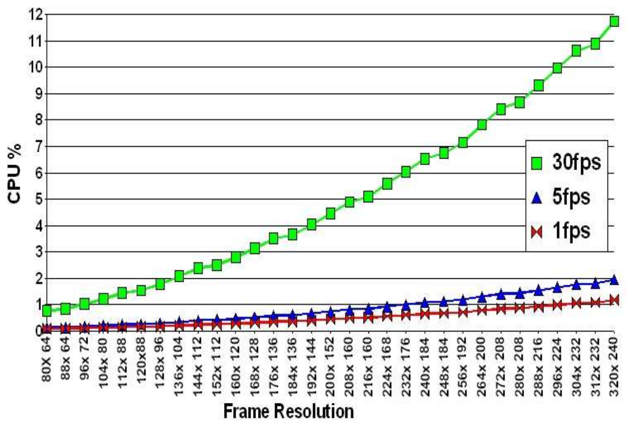


Figure 10.6: CPU utilization during frame processing.

to that of the LCD.

Using the power and performance characterization, we are able to carry out a worst case analysis on the feasibility of the proposed dynamic backlight luminance scaling framework.

Assuming a frame rate of 30 fps and a frame resolution 320x640, the corresponding CPU utilization to these settings is equal to 12%. For 12% of time the core consumes 350 mW, for the remaining time it consumes 10 mW. The average power used by the CPU per second is 50.8 mW. In order to compensate this CPU power, we need to dim the backlight luminance from 255 to 230. This gap is not so wide, therefore we assume that we will be able to compensate the reduced backlight luminance through color modification without a significanr distortion on the image itself.

10.8 Experimental Results

We compared, in terms of power consumption, our hardware/software solution with other approaches. All figures show the power consumption of the LCD display plus the power wasted on CPU. The power consumption of the IPU is not shown because it is constant. Figure 10.7 shows the comparison of our technique against implementation of the same framework with different levels of hardware assistance. The first column in Figure 10.7 shows the power consumed by i.MX31 ADS without any backlight scaling support when it is displaying a video stream: we can consider this as the reference point.

The second column in Figure10.7 shows the power consumption of LCD and CPU during the playback of a video stream with a backlight scaling support similar to our framework but entirely implemented in software. In other

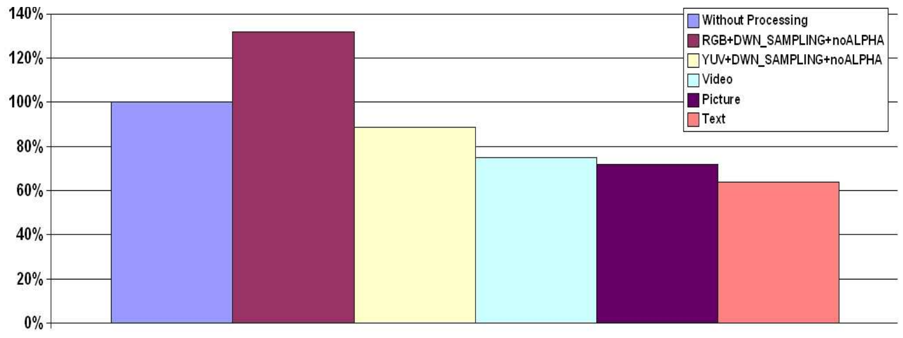


Figure 10.7: Power consumption comparison with different approaches.

words, we implemented the Dynamic Backlight Autoregulation in software without any hardware assistance from the IPU. The video input format is RGB, so CPU has to calculate the brightness histogram of the frame computing a wider set of input data. Instead of computing it on a downsized frame, it is done by downsampling. The color compensation is not done by IPU's alpha-blending, but by the CPU. It has to change pixel by pixel the brightness level of the image, always in RGB format. Figure10.7 shows how this approach is not suitable: compared to the first column (that is no backlight scaling support) we consume 32% more power. The power consumed by the CPU computation introduced by the backlight scaling support is greater than the power saved on the LCD side.

The third column shows the case of software implementation in which we analyze an input video stream in YUV format. In this case we added only a light hardware support: the IPU is used only to convert the video stream from RGB to YUV and back. Results show that this configuration is already suitable, since we are able to save more than 10% of power with respect to the reference case. In this case we experienced greater QoS degradation in terms of final image quality in comparison with our proposed solution, since the distortion is more sensitive to downsampling than downsizing.

The three rightmost columns in Figure10.7 show the power consumption of the system with our hardware/software framework. The different columns show different configurations in terms of frame rate, downsize factor and distortion threshold (see section 4.1). The first column represents the power consumption considering the optimal settings for a video stream application, that is a frame rate of 10 fps, a downsize factor of 0,8 and a distortion threshold of 4%. It shows a power saving of 25% with respect to the reference case (no backlight scaling support). The next column represents the power consumption considering the optimal settings for a still image viewer, with a frame rate of 3 fps, a downsize factor of 1 and a distortion threshold of 3%, and shows

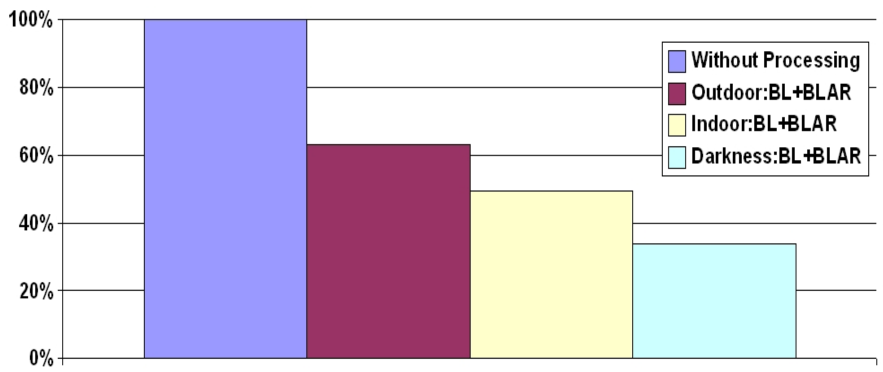
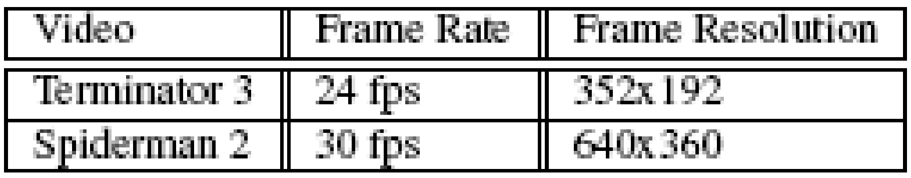


Figure 10.8: Power consumption with different ambient light conditions.



Video	Frame Rate	Frame Resolution
Terminator 3	24 fps	352x192
Spiderman 2	30 fps	640x360

Figure 10.9: Video stream features.

a power saving of 28%. The last column represent the power consumption considering the optimal setting for a text editor, with a frame rate of 1 fps, a downsize factor of 0.25 and a distortion threshold of 10%, and shows a power saving of 36%.

Figure10.8 shows the results of the Ambient-aware Dynamic Backlight Autoregulation integration. In this analysis we took as benchmark a video stream application. The first column is always the reference point with the case of no backlight scaling support. The second column shows the power saving achievable by our framework in an outdoor environment, while the second and the third are namely in an indoor and dark environment. As you can see from the histogram, the darker the ambient luminance, the greater the power saving of our system: it goes from 37% outdoors to 66% in the darkness.

In order to evaluate how our technique works in a real life system, we embedded it in the Video4Linux driver and tested it using the MPlayer video application. We measured the power consumption of the LCD and CPU during the playing of two video: Spiderman 2 and Terminator 3. Table10.9 shows the features of these two video streams in terms of frame rate and resolution.

Figure10.10 and Figure10.11 show the absolute power breakdown achievable during the two video runs. Figure10.8 shows the overall power savings in percentages. Compared to the reference case, with Spiderman 2 savings of 28

Analyzing more in detail Figure 10.10 and Figure 10.11, we notice that with our solution the ratio between power saved on LCD and power consumed on

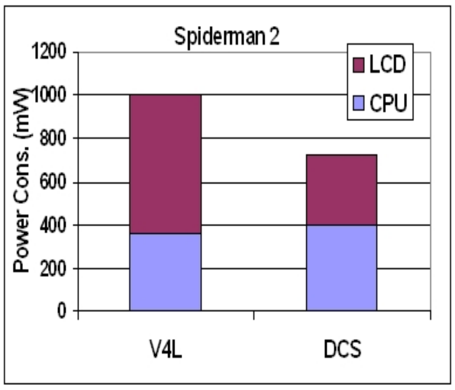


Figure 10.10: Spiderman 2 user case.

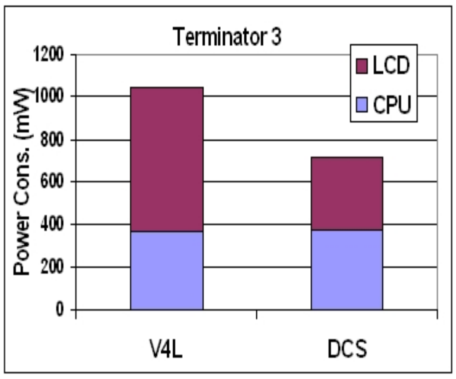


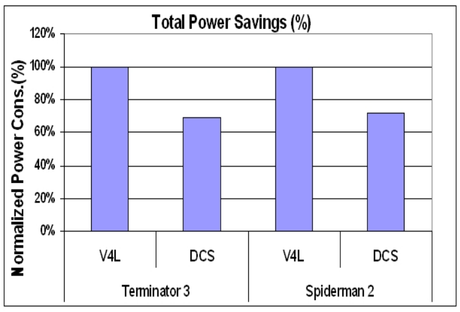
Figure 10.11: Terminator 3 user case.

CPU is different. In Terminator 3 case, CPU consumes 373 mW while LCD 340 mW. In Spiderman 2 case, CPU consumes 397 mW while LCD 324 mW. The CPU consumption is different because the two videos have different features: Spiderman 2 is more computationally challenging for CPU since it has a wider resolution and a faster frame rate (see Table 10.9). The LCD consumption differs because the luminance reduction applied to the LCD backlight, and consequently the power saving achievable, is frame-dependant: the darker the input video stream , the more aggressively the backlight can be scaled.

10.9 Video4Linux Implementation

In this section, the implementation of dynamic backlight scaling technique within Video4Linux is described.

The most important function is ”mxc_v4l2out_streamon”. During this function, all channels and all buffers associated to their streams are instantiated, initialized and enabled. In more detail,, the created channels are: MEM_SDC_BG,



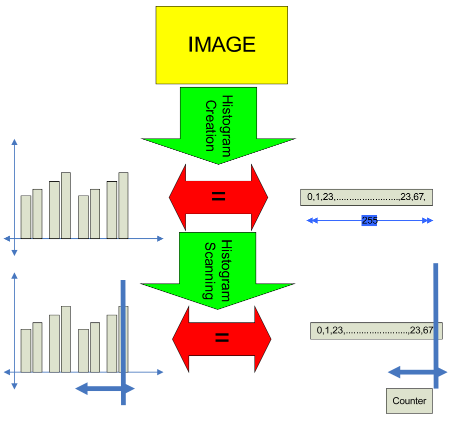


Figure 10.12: creating_histogram_and_scan_histogram function

MEM_SDC_FG, MEM_PRP_ENC_MEM, SDC_FG_FB_FORMAT. For each channel, double buffering both for input and output is used. herein all cases, channels are linked to each other through DMA transfers.

Another important function is "mxc_v4l2out_open", since here all interrupts associated to the pre- and postprocessing are bound to their handlers. The whole synchronization between different IPU submodules and the updating of all buffers are implemented inside these handlers.

The processing phase itself is implemented and triggered inside a handler. In other words, the "mxc_v4l2out_prep_out_irq_handler", that is associated with the end of the preprocessing phase of a frame, calls the fundamental function of the processing phase.

The "DLS_YUV" is the core of the processing step. This function implements the algorithm that calculates the backlight scaling factor. It calls the "creating_histogram_and_scan_histogram" function (see Figure 10.12), which given a frame in YUV format and a distortion level, computes the image histogram and finds the appropriate luminance value that satisfies the distortion constraint.

To create the histogram, this function scans only the Y component of the image and incrementally stores the number of pixels per luminance-value "bucket" in a vector. After that the vector is scanned incrementing a counter. When the counter reaches the distortion level, it represents the luminance dimming value that is required.

10.10 Conclusions

A method for saving significant power in an LCD panel has been demonstrated. The technique, 'Dynamic Luminance Scaling' analyzes and manipulates pixel transmittance values in order to reduce the LCD backlight intensity on-the-fly. It has been demonstrated that the technique is not viable if implemented as software running on the iMX31 CPU since the power consumed in the increased processing overhead exceeds that saved in the LCD backlight. However, by utilizing features of the iMX31 IPU, proprietary to Freescale, significant savings in excess of 20% (¿200mW) are demonstrated using the display under test.

A fully automated LCD backlight regulation scheme has also been demonstrated, further increasing the potential power savings. This sets the 'reference' level of the LCD backlight according to the ambient lighting conditions (measured via a dedicated light sensor, or by multiplexing the on-board camera sensor). This reference level is then scaled dynamically using the DLS scheme described.

Conclusions

Providing support for multimedia applications on low-power mobile devices remains a significant research challenge primarily due to two conflicting aspects: limited HW resources and application high-performance requirements.

Energy efficiency in this kind of platforms can be achieved only via a synergistic hardware and software approach: more effective and QoS-sensitive power management is possible if power awareness and hardware configuration control strategies are tightly integrated with domain-specific middleware services.

The main objective of this PhD research has been the exploration and the integration of a middleware-centric energy management with applications and operating-system. We choose to focus on the CPU-memory and the video subsystems, since they are the most power-hungry components of an embedded system. A second main objective has been the definition and implementation of software facilities (like toolkits, API, and run-time engines) in order to improve programmability and performance efficiency of such platforms.

In this thesis we have contributed tackling some of the numerous open research challenges in the low-power System-on-Chip domain.