Università degli Studi di Bologna

FACOLTÀ DI INGEGNERIA

DOTTORATO DI RICERCA IN INGEGNERIA ELETTRONICA,
INFORMATICA E DELLE TELECOMUNICAZIONI

Ciclo XIX

MULTI PROCESSOR SYSTEM ON CHIP PLATFORM AND STUDYING OF THE BEST ARCHITECTURE AND SOFTWARE SOLUTION FOR AN APPLICATION

Tesi di Dottorato di:

FRANCESCO POLETTI

Relatori :

Chiar. mo Prof. Ing. **LUCA BENINI**

Chiar. mo Prof. Ing. **BRUNO RICCÒ**

Coordinatore:

Chiar. mo Prof. Ing. **PAOLO BASSI**

Settore Scientifico Disciplinare: ING/INF01 Elettronica

Anno Accademico 2006/2007

MULTI PROCESSOR SYSTEM ON CHIP PLATFORM AND STUDYING OF THE BEST ARCHITECTURE AND SOFTWARE SOLUTION FOR AN APPLICATION

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
FRANCESCO POLETTI
born September 07, 1977

March 2007

"This thesis is dedicated to my parents, Erica and all my dear friends. It wouldn't be possible to finish it without all the love I have received."

Acknowledgements

The author wishes to thank: (i) Prof. Luca Benini for his high-level supervision and continuous support, in that no knowledge is possible without a master to follow; (ii) Ing. Davide Brunelli, Claudio Stagni, Federico Angiolini and Martino Ruggiero for his sincere friendship; (iii) All the "unnamed" students that have contributed to this work.

Bologna
March, 2007

Contents

List of Figures

List of Tables

Chapter 1

Introduction

An increasing number of multimedia services (e.g., multi-view video or multiband wireless protocols) are being implemented on embedded consumer electronics thanks to the fast evolution of process technology. These new embedded systems demand complex multi-processor designs to meet their real-time processing requirements while respecting other critical embedded design constraints, such as low energy consumption or reduced implementation size. Moreover, the consumer market is reducing more and more the time-to-market and price [57], which does not permit anymore complete redesigns of such multi-core systems on a per-product basis. Thus, Multi-Processor Systems-on-Chip (MPSoCs) have been proposed as a promising solution for this context, since they are single-chip architectures consisting of complex integrated components communicating with each other at very high speeds [57]. Nevertheless, one of their main design challenges is the fast exploration of multiple hardware (HW) and software (SW) implementation alternatives with accurate estimations of performance, energy and power to tune the MPSoC architecture in an early stage of the design process.

The scope of this dissertation is to explore the MPSoCs design space, explain the work needed to develop a simulation platform and finally shows a real design case. It's divided into two parts, the first one deals with network connectivity at the micro-architectural level and memory architecture. To this purpose, chapters II and III describe the steps for developing a complete on-chip multi-processor simulation platform. Respect to previous work reported in the literature, this simulation environment exhibits very high levels of accuracy in that approximation margins have been reduced to the minimum with respect to real hardware and software architectures. The platform has been described in SystemC [10], a tool that models both hardware

and software by means of a common description language. The developed platform allows cycle-accurate simulation of state-of-the-art multi-processor SoCs, wherein only a few functional units are integrated and therefore a shared bus based communication architecture can still be used. In particular, an AMBA-compliant infrastructure is simulated [36], with the relevant characteristic of allowing the implementation of different arbitration policies as contention resolution schemes for the serialization of simultaneous bus access requests. Chapter IV evaluates the impact of three arbitration policies on system performance (round robin, TDMA and slot reservation) under different traffic patterns on the bus.

Once the simulation platform has been developed, mapping abstract programming models onto tightly power-constrained hardware architectures imposes overheads which might seriously compromise performance and energy efficiency. Therefore, in the second part, we have first performed a comparative analysis of message passing versus shared memory as programming models for single-chip multiprocessor platforms. Our analysis is carried out from a hardware-software viewpoint: we carefully tune hardware architectures and software libraries for each programming model. We analyze representative application kernels from the multimedia domain, and identify application-level parameters that heavily influence performance and energy efficiency. Then, we formulate guidelines for the selection of the most appropriate programming model and its architectural support.

Finally we have studied the tuning of a specific application onto a MPSOC platform. Since high performance chip architectures for biomedical applications is gaining a lot of research and market interest, we have chosen ECG analysis. Our Hardware-Software (HW/SW) Multi-Processor System-on-Chip (MPSoC) design improves upon state-of-the-art mostly for its capability to perform real-time analysis of input data, leveraging the computation horsepower provided by many concurrent DSPs, more accurate diagnosis of cardiac diseases, and prompter reaction to abnormal heart alterations. We have focused on the design methodology to go from the 12-lead ECG application specification to the final HW/SW architecture. We explore the design space by considering a number of hardware and software architectural variants, and deploy industrial components to build up the system.

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

Chapter 2

SoC Co-Simulation

2.1 abstract

We present a co-simulation environment for multiprocessor architectures, that is based on SystemC and allows a transparent integration of instruction set simulators (ISSs) within the SystemC simulation framework. The integration is based on the well-known concept of bus wrapper, that realizes the interface between the ISS and the simulator. The proposed solution uses an ISS-wrapper interface based on the standard gdb remote debugging interface, and implements two alternative schemes that differ in the amount of communication they require. The two approaches provide different degrees of tradeoff between simulation granularity and speed, and show significant speedup with respect to a micro-architectural, full SystemC simulation of the system description.

2.2 Introduction

Today's complex systems-on-chip (SoCs) are usually built from processor based templates, and contain one or more processor cores, with a significant amount of on-chip memory and complex communication busses. Core processors for on-chip integration are often legacy or third-party components, and are viewed as resources. Therefore, designers do not need a detailed description of the processor micro-architecture, but they do require correct functional (behavioral) models and I/O interface descriptors to accurately track the interaction of the core with the rest of the chip. These models should also provide information about the run-time of the software application they execute; such estimates should be reliable enough to cross-validate a design against performance specifications.

Embedded software designers working on processor cores routinely employ cross-development toolkits to validate functionality and assess performance of applications. A minimal cross-development toolkit contains a cross-compiler, a timing-accurate instruction-set simulator (ISS) and a debugger. On the other hand, hardware designers validate their work using hardware-description language (HDL) simulators. The latter are quite inefficient in simulating complex processor cores, because they model their micro-architecture in too much detail.

Designing a complex system-on-chip requires thus a single, integrated hardware-software simulation platform, for both exploration and validation. For this reason, a large number of co-simulation platforms has been developed both by academic groups and EDA vendors [1], [2], [3], [4], [5], [6], [7], [8]. Initially, co-simulation focused establishing a solid link between event-driven hardware simulators and cycle-based ISSs. In the last few years, hardware descriptions and design flows based on C/C++ have gained momentum because of their potential for bridging the gap between hardware and software description languages [9], [10], [11], thanks to the possibility of using the same language for describing software and hardware. In addition, co-simulation becomes easier and more efficient, because the entire system can be simulated within a single simulation engine, eliminating the overhead of communication between different simulators.

SystemC is one of the leading C/C++ design environments: it provides an open-source, free simulation environment and several class packages for specifying hardware blocks and communication channels [10]. Software in SystemC can be specified algorithmically, as a set of functions embedded in SystemC abstract modules. Software modules can communicate among themselves and with hardware components via abstract SystemC communication channels. When software is specified at this level of abstraction, it is very hard to estimate its execution time and analyze its detailed synchronization with hardware.

Excluding the possibility of resorting to cycle-accurate, micro-architectural description of the core, because of its high inefficiency, two approaches are possible. One possibility is that of resorting to a description of the core in SystemC, so that the execution of the software can be modeled consistently with the rest of the system. We will refer to this solution as RTL simulation, to emphasize its cycle-based accuracy. The other option is to simulate the core at a higher abstraction level, by embedding instruction-set simulators within the co-simulation environment.

Most previously published approaches [11], [8], [13], [14], [15] are based on inter-process communication (IPC) and the concept of bus wrapper. The ISS

and the C/C++ co-simulator run as distinct processes on the host system, and they communicate via IPC primitives. The bus wrapper has two key functions:
(i) it ensures synchronization between the system simulation and the ISS;
(ii) it translates the information coming from the ISS into cycle-accurate bus transactions that are exposed to the rest of the system.

Two are the main limitation of these approaches. First, the IPC paradigm is effective when the communication between the ISS and the rest of the system is sparse in time. This is the case when the ISS model includes not just the core but also a significant amount of local memory (e.g., the D-cache), so that communication with the rest of the system is required only for few instructions (e.g., on explicit reads and writes on memory-mapped I/O). Second, most approaches define a proprietary interface between the bus wrappers and the ISS. This choice greatly complicates the integration of new processor cores within the co-simulation framework: the ISS needs to be modified to support the IPC communication primitives defined by the co-simulation system.

This work addresses the two above-mentioned limitations. Our first contribution is an implementation of the IPC interface between bus wrapper and ISS based on the remote debugging primitives of gdb [16]. This can be considered a de-facto standard for IPC, since almost every core processor is provided with a GNU-based software cross-development environment (cross-compiler, ISS and debugger). In this way, any ISS that can communicate with gdb can also become part of a system-level co-simulation environment.

In addition, we address the performance bottleneck created by IPC when the processor interacts very tightly with the rest of the system. We leverage the standardized structure of GNU s instruction-set simulators to develop a small library of functions to be called from within the top module of a legacy GNU's ISS. This top-level module is embedded as a process in the SystemC simulator, and it calls the standard GNU's ISS interface functions, whose implementation is ISS specific. In this way, the ISS is fully embedded in the system simulator executable, and slow interprocess communication is completely eliminated. Results on a system consisting of two processor cores with local and shared memories show the effectiveness of the two proposed co-simulation schemes.

2.3 Co-Simulation Methodology

The proposed co-simulation methodology targets heterogeneous, multi-processor architectures, and is based on the SystemC simulation environment [10]. With respect to the design flow, we assume that the assignment of tasks to either hardware or software (HW/SW mapping) has already been decided. In practice, the multi-processor architectures under analysis consist

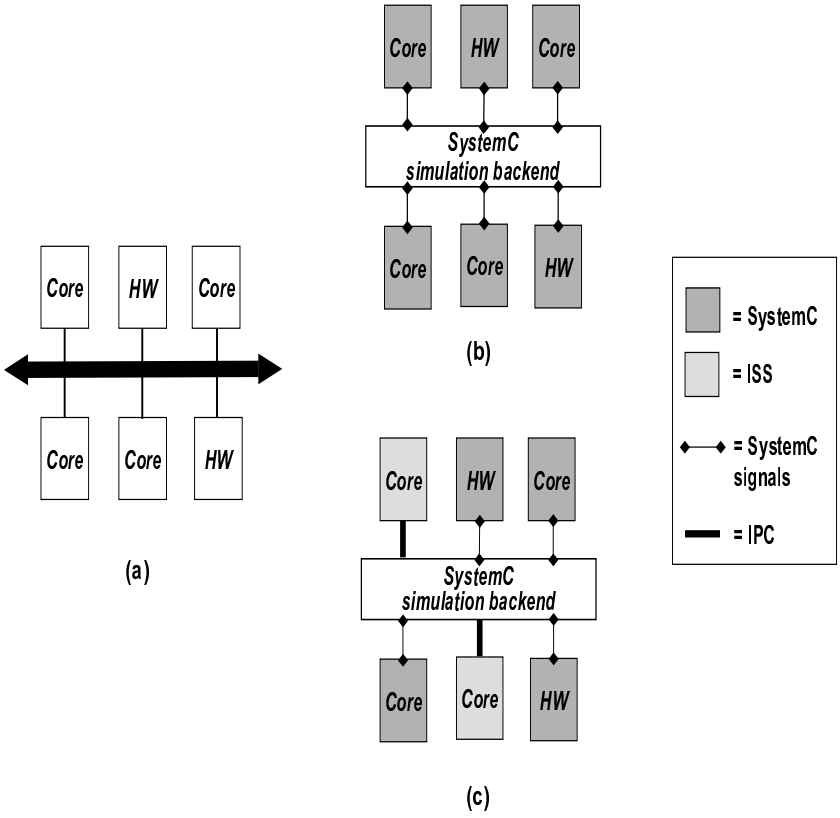


Figure 2.1: Architectural template (a) and simulation alternatives: full SystemC simulation (b) and ISS-SystemC co-simulation (c).

of a set of hardware blocks that implement part of the tasks, and a set of processor cores that execute the other part. Processor mapping is also given, in the sense that specific core platforms have been decided. In this context, the term core identifies a generic programmable resource for which either an ISS or a HDL model is available.

The generic architectural template is shown in Fig. 2.1-(a), where the tasks of the system have been mapped to four cores and two generic hardware block (labeled HW). Fig. 2.1-(b)) shows the case of a full SystemC, cycle-accurate simulation. The co-simulation scheme is depicted in Fig. 2.1-(c)), where, as an example, the SystemC models of some cores are replaced by the corresponding ISSs (the light grey blocks), communicating via IPC with the SystemC simulation back-end. The latter co-simulation scheme is the one followed by most existing approaches [8], [12], [14], that are based on IPC and on the instantiation of bus wrappers; the ISS and the co-simulator run as distinct processes on the host system, and communicate via IPC primitives.

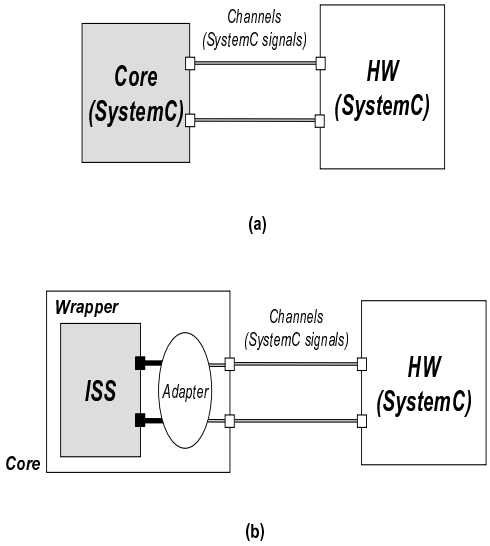


Figure 2.2: Bus wrappers as SystemC modules

The methodology proposed in this work is based on the idea proposed by Semeria and Ghosh, [10]. The use of SystemC allows to eliminate the need of an explicit distinction (from the simulator point of view) between the wrapper and the ISS. The integration of wrappers as SystemC objects allows to restrict the use of IPC just between the bus wrappers and ISS, rather than between bus wrappers (Fig. 2.2). In particular, our methodology overcomes some of the limitations of previous approaches, and has two distinctive features:

- The implementation of the IPC interface between the bus wrapper and the ISS through non-proprietary interface, namely, the remote debugging primitives of the GNU gdb. Compliance of an ISS to this interface becomes then the only constraint for its inclusion in the co-simulation environment. The issue of non-proprietary interfaces for an ISS in co-simulation environments was mentioned in [12], but it was not implemented inside the wrapper abstraction of Fig. 2.2.

- The complete elimination from the co-simulation of the bottleneck of IPCs. This is achieved by adapting the ISS code so that it can be directly embedded as a process in the SystemC simulator. This solution requires the availability of the ISS source code, hence, although most core processors are supported by the GNU cross-development toolkits, it is not viable in the case of proprietary ISSs.

The proposed co-simulation can be realized under two different strategies, that span different degrees of granularities of execution, and are both

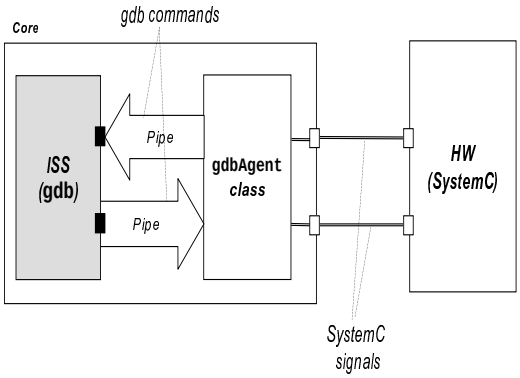


Figure 2.3: Trigger-based co-simulation scheme

based on the integration of wrappers into SystemC. The first strategy, called triggered co-simulation is based on the instantiation of an ad-hoc wrapper that exchanges gdb commands via IPC. The second strategy, called legacy co-simulation implements the scheme that embeds the ISS within the SystemC simulator.

2.4 ISS-SystemC Co-Simulation

2.4.1 Triggered Co-Simulation

The conceptual architecture of the triggered co-simulation approach is depicted in Fig. 2.3. The wrapper consists of a class gdbAgent whose main function is that of executing the gdb and controlling its execution. This class is an extension of a similar class contained in the DDD package, [17], a GNU GUI for the gdb. The constructor of the gdbAgent class first loads and executes the gdb, and creates two UNIX pipes to establish a bidirectional communication channel with gdb, over which conventional gdb commands are exchanged. The class implements then the various methods for driving the execution of the gdb:

- *Quit, Run, Next*: send the corresponding gdb commands, terminated by a newline;
- *setFile*: send the command "file <filename>;"
- *setBreakpoint, setBreakOnCondition*: sets breakpoints on a source line or on some specified condition;

```
void cpuGdb::entry()
{
  cpu.setFileToDebug(application);
  cpu.setTarget("sim");
  cpu.sendCommand("b main\n");
  cpu.setBreakpoint(APPLICATION_FILE,mem_read);
  cpu.setBreakpoint(APPLICATION_FILE,mem_write);
  cpu.sendCommand("b exit\n");

  cpu.Run();
  while (true)
  {
    breakPoint = cpu.contBreak();
    switch(breakPoint)
    {
      case 1: {
        cpu.Next();
        dataBuffer = atoi(cpu.getVariable("dummy"));
        ...
        wait_until(ack.delayed() == true);
        csOut.write(false);
      }
      break;
        ...
    default: ;
    }
    wait();
  }
}
```

Figure 2.4: Typical operation sequence of triggered simulation

- *sendCommand*: sends a gdb command to the ISS;
- *contBreak*: continues after a breakpoint, and return the breakpoint identifier;
- *getVariable, setVariable*: allows to read or modify the value of a variable (correspond to *print var* and *set var=val* commands).

The gdbAgent is compiled within the SystemC environment simulation together with the descriptions of the other modules (possibly with other wrappers), to get a single executable of the whole system description. The granularity of the simulation depends on which gdb command are used to synchronize the execution of the program, and on the system architecture. Fig. 2.4 shows an example of the typical sequence of operations of the triggered scheme, where an object cpu of the gdbAgent class is communicating with the ISS.

Notice the breakpoints in correspondence of two auxiliary functions that expose memory reads and writes. In the case of multiprocessor systems, for instance, synchronization between processors is realized through the access to specific variables in the shared memory. In this case, the coarsest possible granularity is obtained by setting breakpoints in correspondence to reads and writes to those memory cells. The finest granularity is clearly equivalent to

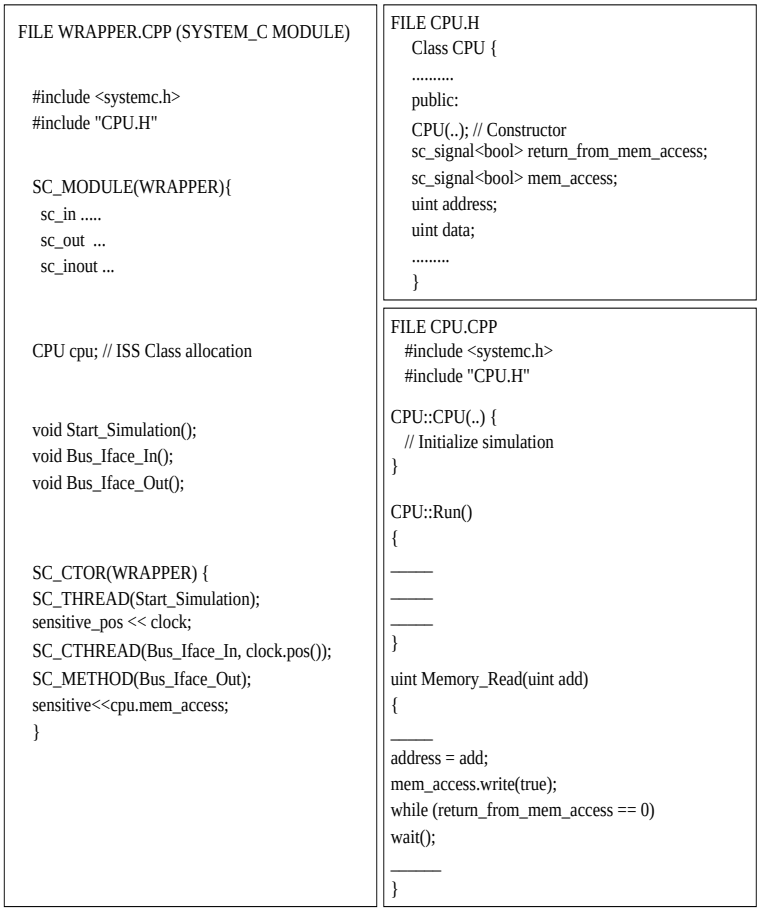


Figure 2.5: SystemC wrapper architecture

tracing instructions step-by-step (via the next command). In this case, however, the overhead due to the IPC becomes non-negligible.

2.5 Legacy ISS Co-Simulation

When the interaction between a processor instance and the rest of the system is very tight, interprocess communication becomes burdensome. In these cases, a tighter link between ISS and SystemC simulations is sought, in an effort to alleviate the speed penalty caused by frequent IPC calls.

An alternative approach to the triggered approach is to completely embed the ISS within the SystemC simulator: in other words, we want to transform the ISS into a C++ class. Upon instantiation of an object of the ISS-class, an instruction set simulation can be started, managed and synchronized with the rest of the system.

More specifically, we define two entities, namely a CPU wrapper SystemC

SC_MODULE WRAPPER and the ISS simulator class CPU. The function of WRAPPER is to instantiate a CPU object, launch the ISS simulation (a method of the CPU object) and synchronize it with the signals from the environment (e.g. from memories and/or peripherals). At the same time, the wrapper implements a virtual socket interface that translates ISS interface events into legal system bus transactions.

The internal organization of the WRAPPER and CPU is outlined next. The CPU class is created starting from a stand-alone ISS (in C or C++). The class declaration is shown in Fig. 2.5. All global variables in the stand-alone ISS must be made internal variables of the CPU class. Locally-scoped variables can remain untouched. Two methods are defined: the CPU constructor and run. The constructor performs all initialization procedures of the standalone ISS, and prepares all data structures required for simulation. The run method performs the simulation. The SystemC SC_MODULE WRAPPER instantiates a CPU object, and initializes it in its constructor. The ISS simulation is started by a dedicated process (a SystemC SC_THREAD), called *Start simulation*. Clearly, if no provisions are made, the run method would run until simulation completion, with no interaction between ISS and its environment. To enable interaction, the code within the run method must be marginally modified. In detail, we change the code around ISS memory and I/O access functions, in such a way that accesses to specific memory or I/O regions can be detected.

An example of this is shown in Fig. 2.5: when an access is detected, some information is made available to the wrapper and execution is suspended with a call to the SystemC wait function. In particular, the SystemC wrapper has to receive information about external memory or I/O addresses, data to write on the bus and the type of bus request (read or write access). Moreover, data read from the bus must be passed back to the ISS. This communication between the ISS and the SystemC wrapper is implemented by allocating the parameters of interest as public variables of the CPU class. In these way, they can be accessed by both sides.

Two other public variables have been used. *Mem_access* triggers the SystemC *Bus_Iface_Out* process, which in turn generates the cycle accurate bus configuration. Return from mem access is the variable watched by the ISS at each recovery from the sleep state, indicating whether the bus access has been completed or not. In this way, the timing penalty for accessing an external memory is taken into account. It is important to note that synchronization between the SystemC time and the ISS simulated time has been implemented. The ISS simulation is suspended by means of wait calls until the SystemC time tracks the simulated time. Only at that time the bus transaction is carried out, thus generating a realistic bus traffic.

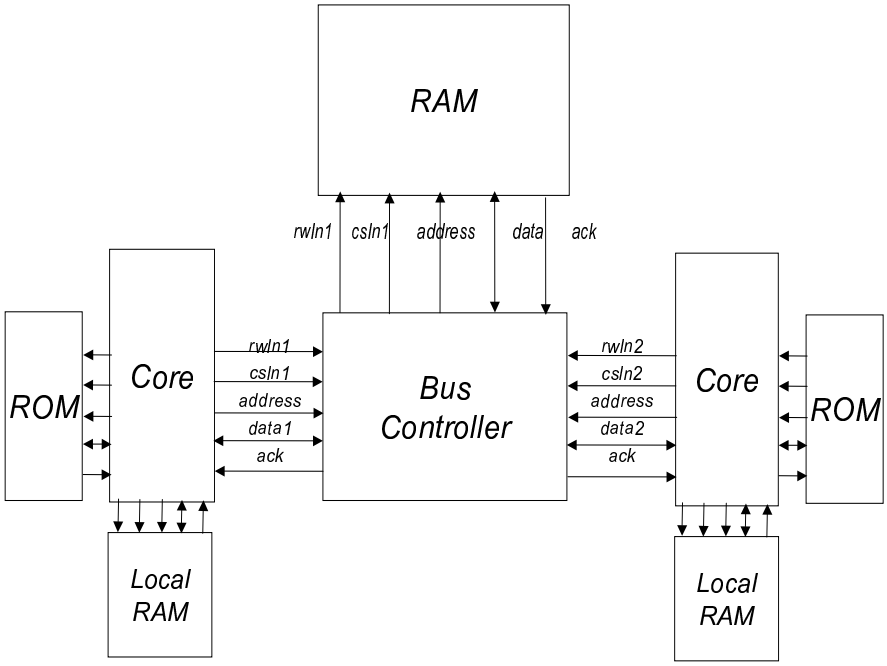


Figure 2.6: Block diagram of the test architecture

2.6 Experimental Results

We have implemented the proposed methodology in the SystemC 2.0 simulation framework, and we have applied it to a system consisting of two core processors accessing to a shared memory through a bus. A block diagram of the system is shown in Fig. 2.6. The bus arbitration mechanism is managed by the module labeled *Bus controller*.

The interface between the bus and the cores consists of five signals: a read/write signal *rwIn*, a chip select *csIn*, an address *addressIn*, the data *data*, and an acknowledge signal from the bus *ACK*, asserted upon completion of a read/write to memory. A similar interface exists between the bus controller and the memory.

Access to the bus is based on a priority mechanism. In order to avoid the chance that one of the processors can be granted the access to the memory for the whole duration of its computation, the bus controller implements a sort of aging mechanism that decreases priorities as the number of memory accesses increases. The application executed by the two processors are stored in a local ROM, and consists of the manipulation (a variant of the computation of a moving average) of an array of integers, executed in a parallel fashion: data

are partitioned in two subsets that are processed concurrently by the two processors.

The processors are synchronized by testing the value of a shared memory cell, used as a semaphore. The availability of a SystemC description of a DLX processor, a simplified version of the MIPS [61], has determined the choice of the target architecture. The relative ISS has been built by means of the GNU cross-compiler (gcc Version 2.95.3) and cross-debugger (gdb Version 5.0) with the MIPS as a target. We have run three different simulation experiments: The first one represents the reference simulation, and consists of a plain SystemC simulation of the architecture of Fig. 2.6.

SIMULATION TYPE	CPU TIME (s)		
	10	100	1000
RTL	9.4	100.3	968
TRIGGERED	2.5	64.7	646.8
LEGACY	0.7	6.6	63.0

Table 2.1: Co-simulation results

All blocks have thus been implemented as SystemC modules, and are synchronized on the same clock. The cores read the respective instructions as binary code from the ROMs, and access the bus according the memory access pattern.

The other two experiments realize the two co-simulation schemes described in Section 3. One experiment uses the triggered approach: the two cores are replaced by two GDBAgent classes, and are driven by the standard GDB interface. Processors are synchronized (i.e., a breakpoint is set) every time a location in the shared memory is modified. In practice, the user issues a conventional GDB *break on* <*condition*> command to synchronize the execution. Notice that only accesses to the shared memory require an explicit interaction via IPC with GDBAgent. Accesses to the local memories always occur through the gdb memory.

The other configuration uses the legacy simulation approach: the two cores are replaced by two CPU classes. The simulation is synchronized, as in the previous case, in correspondence of accesses to locations in the shared memory. Table 2.1 compares the results of the various simulation approaches. The plot reports CPU time, measured on a Pentium II 400 with 256 MB of memory, running Linux Red-Hat 7.2. The table shows three columns 10, 100, and 1000, corresponding to the number of iterations of the algorithm implemented by the application.

The results show that the two co-simulation approaches offer different

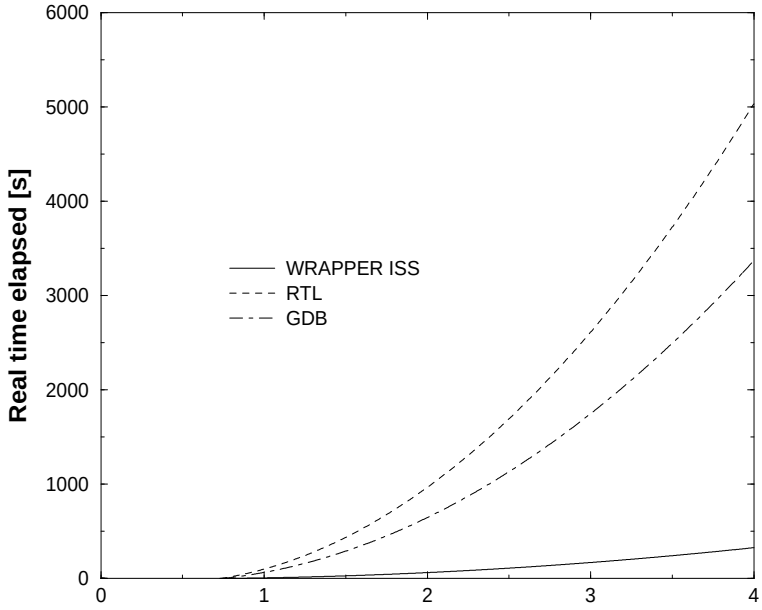


Figure 2.7: Speedup scaling versus number of clock cycles

trade-offs between flexibility and simulation speed. As expected, the legacy approach is much faster, and should always be the choice when the type of synchronization is clearly defined and the target ISS source code is available. The speed-up is more than one order of magnitude with respect to a full SystemC simulation (raw RTL). The triggered approach is slower than legacy simulation, yet still faster than SystemC simulation of a factor of about 2. Notice that, although we labeled the full SystemC simulation as RTL , the implementation is far from being a synthesizable description. As a reference data, the case of 1000 iterations corresponds to the execution of more than 2 million instruction, definitely much faster than a RTL simulation. Therefore, we expect more sizable speed-ups in case of a true RTL simulation of the system.

Fig. 2.7 shows how the speedup scales with respect to the number of iterations of the algorithm (i.e., the number of cycles). Plots have been obtained by spline extrapolation of the data in the table, and shows that, for this specific application, the speed-up increases for larger values of the number of cycles.

2.7 Conclusions

Application of conventional co-simulation paradigms to multi-processor architectures requires efficient mechanisms for the communication between ISSs and the simulation engine. The adoption of a C++-based simulation environment such as SystemC allows to develop effective solution, because the entire system executes within a single simulation environment. In this work, we propose two co-simulation approaches, that are based on the use of a standard interface (namely, the gdb remote debugging interface, supported by most ISS) between the ISS and the wrapper used to link it to the simulation environment. The two proposed solutions provide various degrees of simplification of the ISS/wrapper interface, up to a minimum-overhead scheme that completely removes the need of IPC on the interface, obtained by transparently embedding the ISS within the simulation environment. Simulation results, with respect to a full SystemC simulation of a two-processor test case, shows speed improvements by a factor of 1.5x to 15x, depending on the chosen solution.

Chapter 3

MPARM: a complete Multi-Processor Simulation Platform

3.1 abstract

Technology is making the integration of a large number of processors on the same silicon die technically feasible. These multi-processor systems-on-chip (MP-SoC) can provide a high degree of flexibility and represent the most efficient architectural solution for supporting multimedia applications, characterized by the request for highly parallel computation. As a consequence, tools for the simulation of these systems are needed for the design stage, with the distinctive requirement of simulation speed, accuracy and capability to support design space exploration. We developed a complete simulation platform for a MP-SoC called MP-ARM, based on SystemC as modelling and simulation environment, and including models for processors, the AMBA bus compliant communication architecture, memory models and support for parallel programming. A fully operating linux version for embedded systems has been ported on this platform, and a cross-toolchain has been developed as well. Our MP simulation environment turns out to be a powerful tool for the MP-SOC design stage. As an example thereof, we use our tool to evaluate the impact on system performance of architectural parameters and of bus arbitration policies, showing that the effectiveness of a particular system configuration strongly depends on the application domain and the generated traffic profile.

3.2 Introduction

Systems-on-chips (SoC) are increasingly complex and expensive to design, debug and fabricate. The costs incurred in taking a new SoC to market can be amortized only with large sales volume. This is achievable only if the architecture is flexible enough to support a number of different applications in a given domain. Processor-based architectures are completely flexible and they are often chosen as the back-bone for current SoCs.

Multimedia applications often contain highly parallel computation, therefore it is quite natural to envision Multi-processor SoCs (MPSoCs) as the platforms of choice for multimedia. Indeed, most high-end multimedia SoCs on the market today are MPSoCs [18], [19], [20]. Supporting the design and architectural exploration of MPSoCs is key for accelerating the design process and converging towards the best-suited architectures for a target application domain.

Unfortunately we are today in a transition phase where design tuning, optimization and exploration is supported either at a very high-level or at the register-transfer level. In this chapter we describe a MPSoC architectural template and a simulation-based exploration tool, which operates at the macro-architectural level, and we demonstrate its usage on a classical MPSoC design problem, i.e., the analysis of bus-access performance with changing architectures and access profiles.

To support research for general-purpose multiprocessors in the past, a number of architectural level-multiprocessor simulators have been developed by the computer architecture community [21], [22], [23] for performance analysis of large-scale parallel machines. These tools operate at a very high level of abstraction: their processor models are highly simplified in an effort to speedup simulation and enable the analysis of complex software workloads. Furthermore, they all postulate a symmetric multiprocessing model, which is universally accepted in large-scale, general-purpose multiprocessors.

To enable MPSoC design space exploration, flexibility and accuracy in hardware modeling must be significantly enhanced. Increased flexibility is required because most MPSoC for multimedia applications are highly heterogeneous: they contain various types of processing nodes (e.g. general-purpose embedded processors and specialized accelerators), multiple on-chip memory modules and I/O units, an heterogeneous system interconnect fabric.

These architectures are targeted towards a restricted class of applications, and they do not need to be highly homogeneous as in the case of general-purpose machines. Hardware modeling accuracy is highly desirable because it would make it possible to use the same exploration engine both during archi-

tectural exploration and hardware design.

These needs are well recognized in the EDA community and several simulators have been developed to support SoC design [24], [25], [26], [27], [28]. However, these tools are primarily targeted towards single-processor architectures (e.g. a single processor cores with many hardware accelerators), and their extension toward MPSoCs, albeit certainly possible, is a non-trivial task. In analogy with current SoC simulators, our design space exploration engine supports hardware abstraction level and continuity between architectural and hardware design, but it fully supports multiprocessing.

In contrast with traditional mixed language co-simulators [24], we assume that all components of the system are modeled in the same language. This motivates our choice of SystemC as the modeling and simulation environment of choice for our MPSoC platform. The primary contribution of this chapter is not centered on describing a simulation engine, but on introducing MP-ARM, a complete platform for MPSoC research, including processor models (ARM), SoC bus models (AMBA), memory models, hardware support for parallel programming, a fully operational operating system port (UCLinux) and code development tools (GNU toolchain). We demonstrate how our MPSoC platform enables the exploration of different hardware architectures and the analysis of complex interaction patterns between parallel processors sharing storage and communication resources. Previous work on this topic can be found in [29], [30], [31], [32].

The chapter is organized as follows: Section 2 describes the concepts of the emulated platform architecture and its subsystems (network, master and slave modules), Section 3 shows the software support elements developed for the platform (compiler, peripheral drivers, synchronization, O.S.), Section 4 gives some examples of use of the tool for hardware/software exploration.

3.3 Multiprocessor simulation platform

Integrating multiple ISSs in a unified system simulation framework entails several non-trivial challenges, such as the synchronization of multiple CPUs to a common time base, or the definition of an interface between the ISS and the simulation engine.

The utilization of SystemC [33] as back-bone simulation framework represents a powerful solution for embedding ISSs in a framework for efficient and scalable simulation of multiprocessor SoCs. Besides the distinctive features of modeling software algorithms, hardware architectures and SoC or system level designs interfaces, SystemC functionalities make it possible to plug an ISS into the simulation framework as a system module, activated by the common sys-

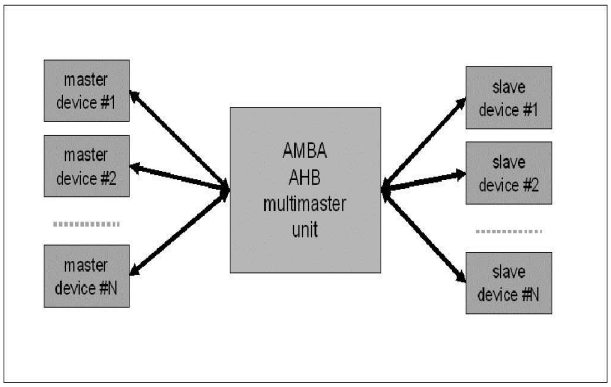


Figure 3.1: System architecture

tem clock provided to all of the modules (not physical clock).

SystemC provides a standard and well defined interface for the description of the interconnections between modules (ports and signals). Moreover, among the advantages of C/C++ based hardware descriptions, there is the possibility of bridging the hardware/software description language gap [34].

SystemC can be used in such a way that each module consists of a C/C++ implementation of the ISS, encapsulated in a SystemC wrapper. The wrapper realizes the interface and synchronization layer between the instruction set simulator and the SystemC simulation framework: in particular, the cycle-accurate communication architecture has to be connected with the coarser granularity domain of the ISS.

The applicability of this technique is not limited to ISSs, but can be extended to encapsulate C/C++ implementations of system blocks (such as memories and peripherals) into SystemC wrappers, thus achieving considerable speed-ups in the simulation speed. This methodology trades-off simulation accuracy with time, and represents an efficient alternative to the full SystemC description of the system modules (SystemC as an hardware description language) at a lower abstraction level. This former solution would slow-down the simulation, and for complex multiprocessor systems this performance penalty could turn out to be unacceptable.

A co-simulation scenario can also be supported by SystemC, where modules encapsulating C++ code (describing the simulated hardware at a high level of abstraction, i.e. behavioural) coexist with modules completely written in SystemC (generally realizing a description at a lower level of abstraction). In this way, performance versus simulation accuracy can be tuned and differentiated between the modules.

Based on these guidelines, we have developed a multiprocessor simulation framework using SystemC 1.0 as simulation engine. The simulated system currently contains a model of the communication architecture (compliant with the

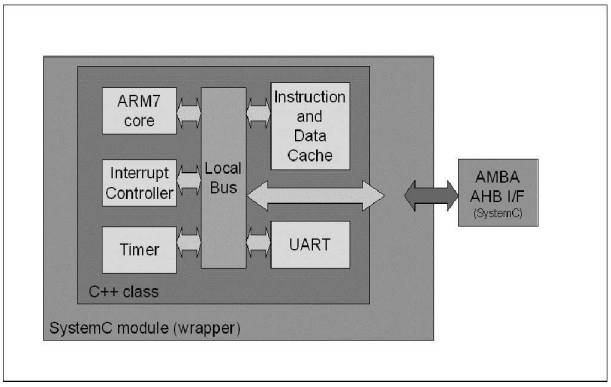


Figure 3.2: Processing module architecture

AMBA bus standard), along with multiple masters (CPUs) and slaves (memories) (Fig. 3.1). The intrinsic multi-master communication supported by the AMBA protocol has been exploited by declaring multiple instances of the ISS master module, thus constructing a scalable multiprocessor simulator.

3.3.1 Processing modules

The processing modules of the system are represented by cycle accurate models of cached ARM cores. The module (Fig. 3.2) is internally composed of the ARM CPU, the first-level cache and peripheral (UART, timer, interrupt controller) simulators written in C++.

It was derived from the open source cycle accurate SWARM (software ARM) simulator[18] encapsulated in a SystemC wrapper. The SWARM simulator is entirely written in C++. It emulates an ARM CPU and is structured as a C++ class which communicates with the external world using a Cycle function, which executes a clock cycle of the core, and set of variables in very close relation to the corresponding pins of a real hardware ARM core. Along with the CPU, a set of peripherals is emulated (timers, interrupt controller, UART) to provide support for an Operating System running on the simulator.

The cycle-level accuracy of the SWARM simulator simplifies the synchronization with the SystemC environment (i.e. the wrapper module), especially in a multiprocessor scenario, since the control is returned to the main system simulator synchronizer (SystemC) at every clock cycle [35].

The interesting thing about ISS wrapping is that with relatively little effort, other processor simulators can be embedded in our multiprocessor simulation back-bone (e.g. mips). Provided they are written in C/C++, their access requests to the system bus need to be trapped, so to be able to make the communication extrinsic and generate the cycle accurate bus signals in compliance with the communication architecture protocol. Moreover, the need for a syn-

chronization between simulation time and ISS simulated time arises only when the ISS to be embedded has a coarse time resolution, i.e. when it does not simulate each individual processor clock cycle.

Finally, the wrapping methodology determines negligible communication overhead between the ISS and the SystemC simulation engine, because the ISS does not run as a separate thread and consequent communication primitives are not required, that would otherwise become the bottleneck with respect to the simulation speed.

3.3.2 AMBA bus model

AMBA is a widely used standard defining the communication architecture for high performance embedded systems [36]. Multi-master communication is supported by this back-bone bus and requests for simultaneous accesses to the shared medium are serialized by means of an arbitration algorithm.

The AMBA specification includes an advanced high-performance system bus (AHB), and a peripheral bus (APB) optimized for minimal power consumption and reduced interface complexity to support connection with low-performance peripherals. We have developed a SystemC description only for the former one, given the multi-processor scenario we are targeting. Our implementation supports the distinctive standard-defined features for AHB, namely burst transfers, split transactions and single-cycle bus master handover.

The model has been developed with scalability in mind, so to be able to easily plug-in multiple masters and slaves through proper bus interfaces. Bus transactions are triggered by asserting a bus request signal. Then the master waits until bus ownership is granted by the arbiter: at that time, address and control lines are driven, while data bus ownership is delayed by one clock cycle, as an effect of the pipelined operation of the AMBA bus. Finally, data sampling at the master side (for read transfers) or slave side (for write transfers) takes place when a ready signal is asserted by the slave, indicating that on the next rising edge of the clock the configuration of the data bus can be considered stable and the transaction can be completed.

Besides single transfers, four, eight and sixteen-beat bursts are defined in the AHB protocol too. Unspecified-length bursts are also supported. An important characteristic of AMBA bus is that the arbitration algorithm is not specified by the standard, and it represents a degree of freedom for a task-dependent performance optimization of the communication architecture. A great number of arbitration policies can be implemented in our multiprocessor simulation framework by exploiting some relevant features of the AMBA bus.

For example, the standard allows higher priority masters to gain ownership of the bus even though the master which is currently using it has not completed yet. This is the case of the early burst termination mechanism, that comes into play whenever the arbiter does not allow a master to complete an ongoing burst. In this case, masters must be able to appropriately rebuild the burst when they next regain access to it.

Our multiprocessor simulation platform allows design space exploration of arbitration policies, and to easily derive the most critical parameters determining the performance of the communication architecture of a MP-SoC. This capability of the simulation environment is becoming of critical importance, as the design paradigm for SoC is shifting from device centric to interconnect centric [20]. The efficiency of a certain arbitration strategy can be easily assessed for multiple hardware configurations, such as number of masters, different master characteristics (e.g. cache size, general purpose versus application specific, etc.).

3.3.3 Memory sub-system

The system is provided with two hierarchies of memories, namely cache memory and main memory. The cache memory is contained in the processing module and is directly connected to the CPU core through its local bus. Each processing module has its own cache, acting as a local instruction and data memory; it can be configured as a unified instruction and data cache or as two separate banks of instruction and data caches. Configuration parameters include also cache size, line length and the definition of non cacheable areas in the address space.

Main memory banks reside on the shared bus as slave devices. They consist of multiple instantiations of a basic SystemC memory module. Each memory module is mapped on its reserved area within the address space; it communicates with the masters through the bus using a request-ready asynchronous protocol; the access latency - expressed in clock cycles - is configurable.

3.3.4 Multiprocessor synchronization module

In a multiprocessing system there is the need for an hardware support for process synchronization in order to avoid race conditions when two or more processes try to access the same shared resource simultaneously. The support for mutual exclusion is generally provided by ad hoc non-interruptible CPU instructions, such as the test_and_set instruction.

In a multiprocessor environment the presence of non-interruptible instructions must be combined with external hardware support in order to obtain

mutual exclusion of shared resources between different processors. We have equipped the simulator with a bank of memory mapped registers which work as hardware semaphores. They are shared among the processors and their behavior is similar to that of a shared memory, with the difference that when one of these 32 bit registers is read, its value is returned to the requester, but at the same time the register is automatically set to a predefined value before the completion of the read access. In this way a single read of one of the registers works as an atomic test_and_set function. This module is connected to the bus as a slave and its locations are memory mapped in a reserved address space.

3.4 Software support

The cross-compilation toolchain includes the GNU gcc-3.0.4 compiler for the ARM family of processors ad its related utilities, compiled under Linux. The result of the compilation and linking step is a binary image of the memory, which can be uploaded into the simulator.

3.4.1 Operating system support: uCLINUX

Hardware support for booting an operating system has been provided to the simulator through the emulation of two basic peripherals needed by a multitasking O.S.: a timer and an interrupt controller. An additional UART I/O device allows to display startup, error and debug information on a virtual console. Linux-style drivers have been written for these devices, running under the linux 2.4 kernel.

The kernel version ported onto the emulation platform consists of a reduced version of linux (uClinux) for embedded systems without memory management unit support [38]. Our simulation platform allows to boot multiple parallel uCLinux kernels on independent processors and to run benchmarks or interactive programs, using the UART as an I/0 console.

3.4.2 Support for multiple processors

The software support for multiprocessors includes the initialization step and synchronization primitives, together with some modifications of the memory map. When a processor performs an access to the memory region where it expects to find the exception vectors, the address has been shifted to a different region in the main memory, so that each processor can have its own distinct exception table. The result is a virtual memory map specific for each processor (Fig. 3.3), which must not be confused with a general purpose memory management support.

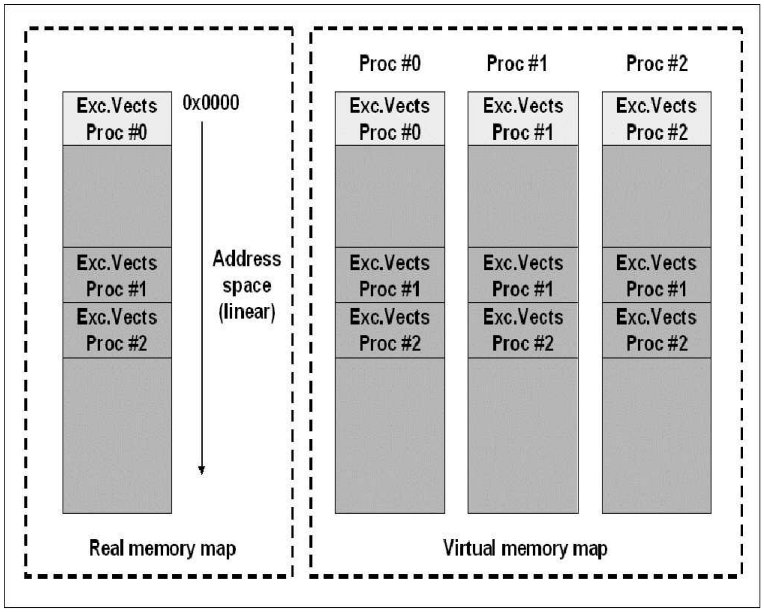


Figure 3.3: Memory map

Having its own reset vector, each processor can execute its own startup code independently on the others. Each processor initializes its registers (e.g. stack pointer) and private resources (timers, interrupt controllers). Shared resources are initialized by a single processor while the others wait using a semaphore synchronization method. At the end of the initialization step, each processor branches to its own main routine (namely main0, main1, main2, etc.).

The linker script is responsible for the allocation of the startup routines and of the code and data sections of the C program. Synchronization software facilities includes definitions and primitives to support the hardware semaphore region (multiprocessor synchronization module) at C programming level. The routines consists of a blocking test_and_set function, of a non-blocking test function and of a free function.

3.5 Experimental results

Our simulation environment can be used for different kinds of design exploration, and this section will give some examples thereof. To this purpose, we used the aforementioned software toolchain to write some benchmark programs for a two-processors system with different levels of data interaction between the two processors. Fig. 3.4 shows the common system architecture configuration used for the examples.

Two processing ARM modules are connected to the AMBA bus and act as masters, and two identical memory modules are connected as slaves and can

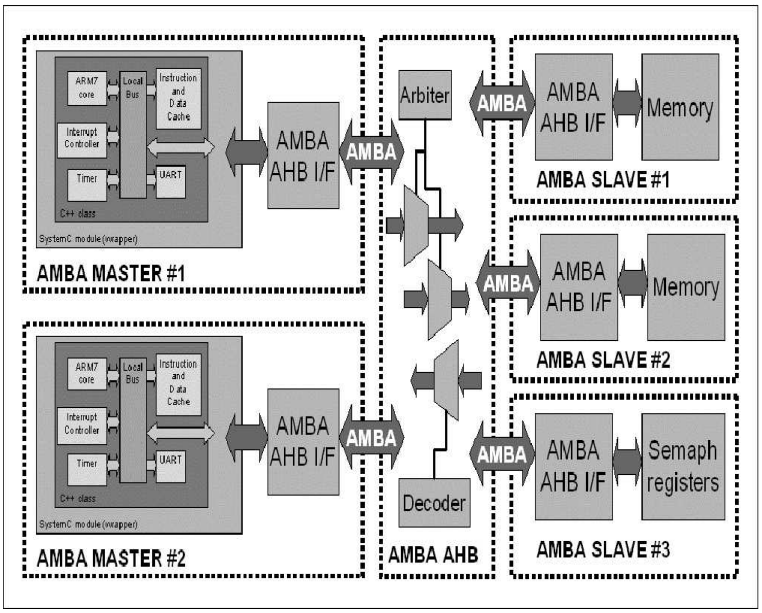


Figure 3.4: System architecture for benchmark examples

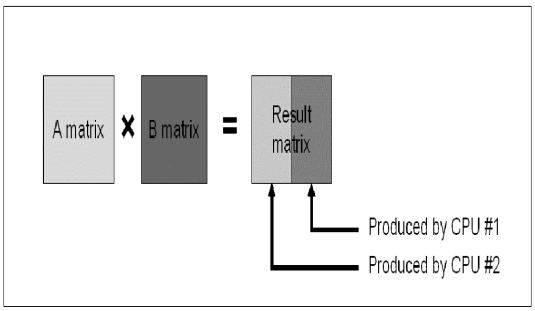


Figure 3.5: Matrix multiplication

be accessed by both processors. The third slave module is the semaphore unit, used for synchronization in one of the following benchmark programs.

3.5.1 Benchmark description

1. **same data set program (shared data source)**
   The two processors execute the same algorithm (matrix multiplication) on the same data source. In this program half the result matrix is generated by the first processor while the other half is generated by the other processor (Fig. 3.5). The two processors share the source data (the two matrixes that have to be multiplied), but there are no data dependencies between them, so there is no need to use synchronization functions between the processors.

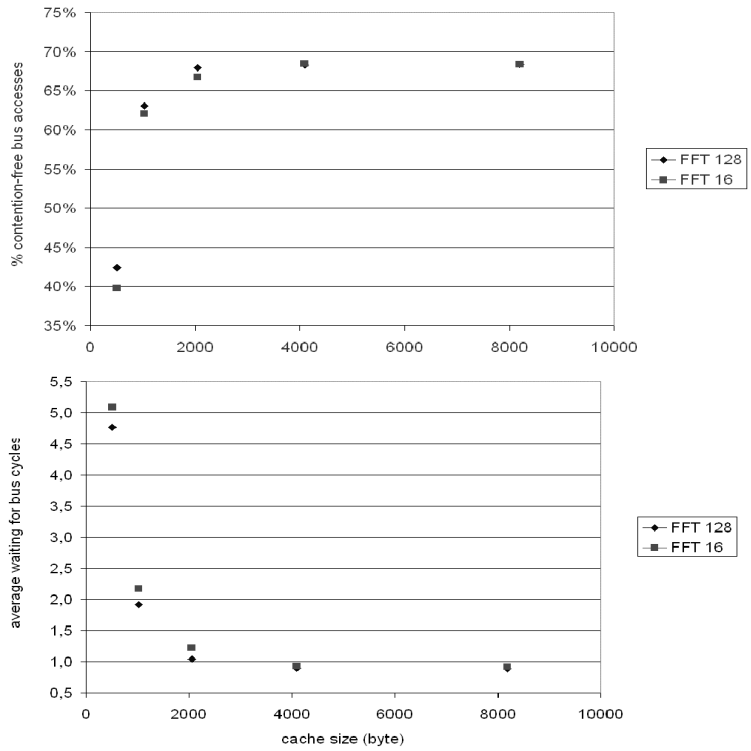


Figure 3.6: (a) Contention free bus accesses versus cache size (b) Average waiting time for bus access versus cache size

2. **data dependent program (producer-consumer algorithm)**
   The first processor execute a one-dimensional N-size integer FFT on a data source stream while the second execute a one-dimensional N-size integer IFFT on the data produced by the first processor. For each N-size FFT block completed, a dedicated semaphore is released by the first CPU before initiating data elaboration of the subsequent block. The second CPU, before performing the IFFT on a data block will check its related semaphore and will be locked until data ready will be signaled.

3.5.2 Architectural exploration

In this example we show the results obtained running the above-mentioned benchmarks and varying architectural or program parameters. The explored parameters are two, one related to the system architecture, cache size, and the other related to the program being executed, FFT size (which affects data locality). The FFT performed on an N-size block will be hereafter indicated as ”FFT N”.

In Fig. 3.6 and Fig. 3.7 we graphically illustrate the results relative to contention-free bus accesses (percentage of times a CPU is immediately

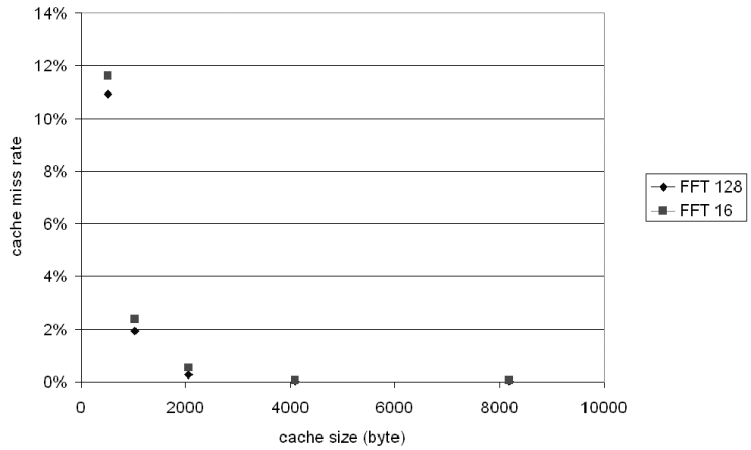


Figure 3.7: Cache miss rate versus cache size

granted the bus against its access requests, with respect to the total number of bus access requests), average waiting time before gaining bus ownership (this delay is a side-effect of the arbitration mechanism and of the serialization of bus requests), average cache miss rate for the two processors.

3.6 Conclusions

We have developed a complete platform for the simulation of a MP-SoC, allowing investigation in the parameter space (related to the architecture configuration or to the protocols) to come up with the most efficient solution for a particular application domain. Our platform makes use of SystemC as simulation engine, so that hardware and software can be described in the same language, and is based on an AMBA bus compliant communication architecture. ARM processors act as bus masters (like in commercial high-end multimedia SoCs), and the simulation platform includes memory modules, synchronization tools, and support for system software (porting of the uClinux OS and development of a cross-toolchain.) We have shown examples of applications wherein our simulation environment is used to explore some design parameters, namely cache parameters and bus arbitration policies. The applications involve data-independent or data-dependent tasks running on different ARM CPUs sharing the main memory through a common AMBA bus. The examples show how to derive important metrics (cache size, average waiting time for accessing the bus since the request is asserted, etc.) that heavily impact system performance, proving its effectiveness in supporting the design stage of a multi-processor system-on-chip.

Chapter 4

Performance Analysis of Bus Arbitration Schemes

4.1 abstract

As technology scales toward deep submicron, the integration of a large number of IP blocks on the same silicon die is becoming technically feasible, thus enabling large-scale parallel computations, such as those required for multimedia workloads. The communication architecture is becoming the bottleneck for these multiprocessor Systems-on-Chip (SoC), and efficient contention resolution schemes for managing simultaneous access requests to the shared communication resources are required to prevent system performance degradation. The contribution of this work is to analyze the impact on multiprocessor SoC performance of different bus arbitration policies under different communication patterns, showing the distinctive features of each policy and the strong correlation of their effectiveness with the communication requirements of the applications. Beyond traditional arbitration schemes such as round robin and TDMA, another policy is considered that periodically allocates a temporal slot for contention-free bus utilization to a processor which needs fixed predictable bandwidth for the correct execution of its time-critical task. The results are derived on a complete and scalable multiprocessor SoC simulation platform based on SystemC, whose software support includes a complete embedded multiprocessor OS (RTEMS). The communication architecture is AMBA compliant, and we exploit the flexibility of this multi-master commercial standard, which does not specify the arbitration algorithm, to implement the explored contention resolution schemes.

4.2 Introduction

Deep submicron technologies are making the integration of a large number of IP blocks on the same silicon die technically feasible. As a consequence, several heterogeneous cores can be combined through sophisticated communication architectures on the same integrated circuit, leading to the development of flexible hardware platforms able to accommodate highly parallel computation. The application domain of these Systems-on-Chip (SoC) includes mobile terminals (e.g. for multimedia applications), automotive, set-top-boxes, game processors, etc. [39].

The SoC design paradigm relies heavily on re-use of intellectual property cores (IP cores), enabling designers to focus on the functionality and performance of the overall system. This is possible if the IP cores are equipped with a highly optimized interface for their plug-and-play insertion into the communication architecture. To this purpose, the Virtual Socket Interface Alliance (VSIA) represents an attempt to set the characteristics of this interface industry-wide, thus facilitating the match of pre-designed software and hardware blocks from multiple sources [41] [40].

The most widely adopted interconnect architecture for the SoC IP blocks is bus-based, and consists of shared communication resources managed by dedicated arbiters that are in charge of serializing access requests. This architecture usually employs hierarchical buses, and tends to distinguish between high performance system buses and low complexity and low speed peripheral buses. Many commercial on-chip architectures have been developed to support the connection of multiple bus segments in arbitrary topologies, providing at the same time a moderate degree of scalability: Wishbone [44], AMBA [43] and CoreConnect [42] are relevant examples.

As the complexity of SoCs increases, the communication architecture becomes the performance bottleneck of the system. The performance of multiprocessor systems depends more on the efficient communication among processors and on the balanced distribution of the computation among them, rather than on pure CPU speed. For integration levels in the order of hundreds of processors on the same SoC, the most efficient and scalable solution will be the implementation of micronetworks of interconnects [46], but below that limit bus-based communication architectures remain the reference solution for state-of-the-art multiprocessor systems because of the lower design effort and hardware cost. This forces designers to push the performance of these architectures to the limit, within the architectural degrees of freedom made available by existing commercial bus standards.

The arbitration process plays a crucial role in determining the performance

of the system, as it assigns the priorities with which processors are granted the access to the shared communication resources. The increasing integration levels of a SoC translate to an increase of contention among the processing elements for the bus, and this might lead to the violation of real-time constraints and more in general to performance degradation. An efficient contention resolution scheme is therefore required to support real-time isochronous data flows associated with networking and multimedia data streams.

4.3 Contribution of this work

An effective bus arbitration policy should satisfy several requirements: (i) enable fast, high-priority communication, while avoiding starvation of low priority transactions; (ii) provide fine-grained control of the communication bandwidth allocated to individual system components; (iii) reduce sensitivity of system performance to variations of the communication pattern induced by an application on the bus. Traditional arbitration policies used both by centralized and by distributed arbiters to address the bus contention problem in multi-master SoCs include priority based selection, round robin and time division multiple access (TDMA). More advanced arbitration algorithms have been also proposed [45].

The main contribution of this work is to point out the correlation between the effectiveness of an arbitration policy and the traffic pattern induced on the bus by the communication requirements of an application. In particular:

- Beyond investigating how system performance is affected by conventional arbitration policies such as round robin and TDMA, we extend our analysis to a ”slot reservation” policy, wherein a temporal slot for contention-free bus utilization is periodically reserved to a specific processor which needs a guaranteed bandwidth for the correct execution of its task. During the inter-slot time, all other processing elements compete for accessing the bus in a round-robin fashion.

- We show that the optimal contention resolution scheme is not unique, and we analize three case studies which are representative of different communication patterns where the considered arbitration policies compare differently. For each scenario, we use a performance metric indicating how efficiently an application (or a set of applications) running on top of a multiprocessor platform is executed, and we show the impact of the arbitration policies on this metric.

- We provide experimental results obtained by means of extensive simulations on a complete and scalable multiprocessor SoC simulation plat-

form, hereafter denoted as MPARM [47]. The simulation backbone is SystemC, that allows the description of both hardware and software in a common language (C++). The hardware platform consists of a scalable number of cycle-accurate ARM instruction set simulators, embedded into SystemC wrappers implementing the interface between the cores and an AMBA-based communication architecture.

- We developed a complete software development and run-time support infrastructure for MPARM, including a complete port of an embedded multiprocessor operating system (RTEMS).

The chapter is structured as follows: Section 4.4 provides an overview of previous work, Section 4.5 describes bus arbitration policies, Section 4.6 summarizes the key features of the AMBA bus specification, Section 4.7 describes the multiprocessor simulation platform used for our experiments, Section 4.8 presents experimental results and Section 4.9 concludes the chapter.

4.4 Previous work

Communication architectures defined by commercial standards always provide a certain degree of flexibility in arbitration policies. This allows end users or SoC manifacturers to tailor the hardware architecture to the particular application domain of interest.

The CoreConnect interconnect architecture from IBM makes use of a fixed priority arbiter, but the priority fairness is programmable [42]. Therefore designers must analyze the application and determine the priorities among devices. Up to 8 masters on the same system bus can be managed, and address pipelining is supported.

AMBA specification from ARM [43] shares many characteristics with CoreConnect, e.g. the pipelined operation of the bus and bus segmentation and bridging to support communication diversity. However, the arbiter implementation is more flexible: although the arbitration protocol is fixed, any arbitration policy can be implemented depending on the application requirements. Wishbone from Silicore Corp.[44] is another bus specification wherein arbitration is defined by the end-user. Silicon Backplane from Sonic Inc. [48] is a solution for communication among IP cores that guarantees fixed bandwidths and latencies by means of TDMA-based arbitration.

A significant effort has been recently devoted to enabling the effective design of multicore SoC starting from pre-designed and pre-verified IP blocks. An overview of design methodologies and tools proposed to address the problem is provided by [53] [54]. The communication architecture is a key point of

the design stage, and the final goal of many works is to extract the communication requirements from the application and to map them to the underlying communication architectures by looking for the solution that enables to meet application-perceived performance constraints [50]. To this purpose, a design space exploration technique for SoC communication architectures is proposed in [49].

A comparison between SoC bus architectures is made in [51], where selection guidelines for such architectures are also provided. [52] proposes an adaptive SoC communication infrastructure that can be easily reconfigured as application-level communication pattern changes: it provides support for compile-time predicted inter-node communication.

Finally, a new high performance architecture for SoC design is presented in [45], called ”Lotterybus”. It consists of a randomized arbitration algorithm implemented in a centralized ”lottery manager”, which collects requests for bus ownership from multiple masters. The manager probabilistically chooses one of the contending masters which is granted the bus for one or more cycles. The performance of this policy is compared to that of conventional communication architectures for different traffic classes.

The main shortcoming of previous explorative work in bus arbitration is that system functionality is taken into account in a highly abstract fashion. Most previous works employ stochastic traffic generators as bus masters, while others use high-level functional models for software tasks, that do not account for non-ideality of software execution on a target processor (e.g., instruction misses, operating system overhead). Our single-chip multiprocessor simulation platform is cycle accurate both at the bus transaction level and at the software execution level, and fully functional applications and OS are executed without any abstraction or simplification (no instructions are emulated on the simulation host). The approximation margins in our explorative analysis are therefore reduced to a minimum.

4.5 Contention resolution schemes

In this section we briefly present and discuss the key features of the arbitration policies analyzed in the remainder of the chapter.

4.5.1 Round-robin

A round-robin arbitration policy is a token passing scheme wherein fairness among masters is guaranteed, and no starvation can take place (in constrast with a static fixed priority scheme) [56]. In each cycle, one of the masters (in

round-robin order) has the highest priority for access to a shared resource. If the token-holding master does not need the bus in this cycle, the master with the next highest priority who sends a request can be granted the resource. The advantages of round-robin are twofold:

- Unused time slots are immediately re-allocated to masters which are ready to issue a request, regardless to their access order. This reduces bus under-utilization in comparison with a statically fixed slot allocation, that might grant the bus to a master which is not going to carry out any communication.
- The worst-case waiting time for the bus access request of a master is reliably predictable (being proportional to the number of instantaneous requests minus one), even though the actual waiting time is not. The uncertainty on the actual bandwidth that can be granted to a master is the major drawback of this scheme.

4.5.2 TDMA

A time division multiple access scheme is based on the fixed allocation of a slot to each master, so that each of them is guaranteed fixed and predictable bandwidth. Unfortunately, high priority communications in a TDMA-based architecture may incur significant latencies, because the performance provided by this scheme strongly depends on the time-alignment of communication requests and slot allocation, and therefore on the probability of dynamic variations of the request patterns.

4.5.3 Slot reservation

This arbitration policy can be seen as a limit case of TDMA, in that only one master is periodically allocated a slot for the contention-free access to the bus. For the inter-slot time, we decided to manage the contention among the remaining masters in a round-robin fashion. Although this is not a conventional scheme for SoC communication architectures, we propose this policy to combine the advantages of the above mentioned schemes: one master is given a priviledge in the competition for bus access (in terms of guaranteed fixed bandwidth), while all other masters can contend for the shared communication resource avoiding the risk of starvation.

The highest priority master (the one to which the slot is allocated) can therefore complete its transfers without incurring contention-related delays, which are likely to considerably increase as the number of competing masters increases. This translates to a performance degradation for the lower priority

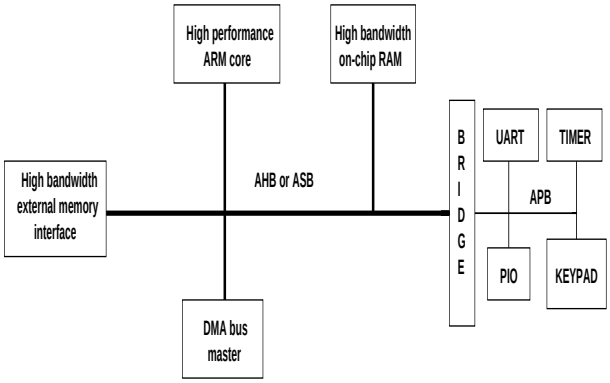


Figure 4.1: A typical AMBA system

masters (those managed in a round-robin fashion), which are excluded from bus access during the slot allocation. The effectiveness of this scheme is tightly related to the ratio between the performance improvement of the highest priority master and the performance degradation suffered by the lowest priority ones, which must be as high as possible.

4.6 AMBA bus

The Advanced Microcontroller Bus Architecture (AMBA) defines an on-chip communication standard for designing high-performance multi-master SoCs. Three distinct buses are defined within the AMBA specification, as illustrated in Fig. 4.1: (i) the Advanced High-Performance Bus (AHB), which is the highly optimized system backbone bus; (ii) the Advanced System Bus (ASB), an alternative system bus used whenever less aggressive performance is required; (iii) the Advanced Peripheral Bus (APB), which is a low complexity and low power bus for communication with general purpose peripherals. The system and the peripheral bus are connected to each other by means of a bridge which reduces global wires load capacitances and hence switching power consumption.

4.6.1 Arbitration protocol

In this work we will focus on AHB, which exhibits high performance features such as support for multiple masters and multiple slaves, for pipelined bus operation and burst transfers, as well as for split transactions.

As already mentioned, AMBA defines the arbitration protocol but it does not define the contention resolution policy. A bus master requests to access the bus by asserting a HBUSREQ signal, as illustrated in Fig. 4.2. The arbiter observes the different simultaneous requests and grants the bus to the highest priority master by asserting its HGRANT signal. The master effectively gains

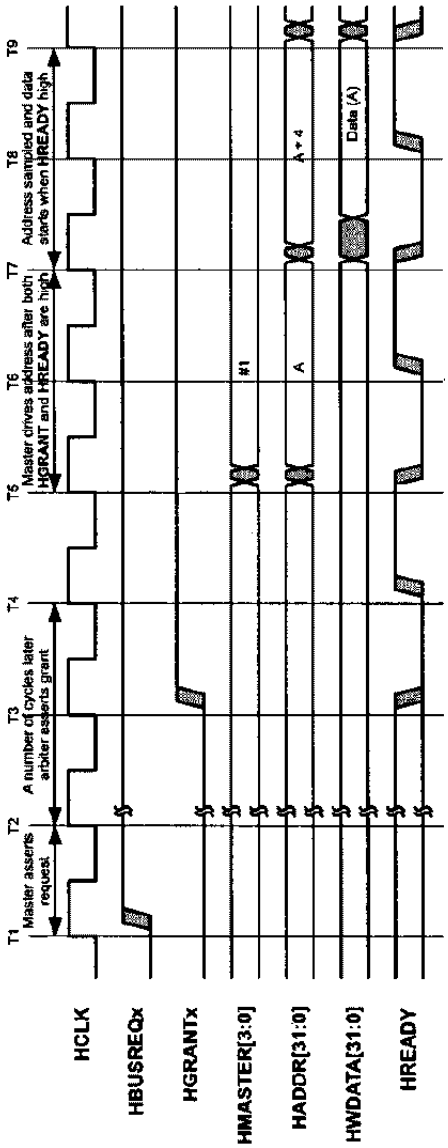


Figure 4.2: Bus handover within the AMBA specification

control of the address bus when both the HGRANT and the HREADY signals (indicating that the last transfer has completed) are sampled high. The ownership of the data bus is delayed with respect to the ownership of the address bus, thus allowing pipelined operation.

4.6.2 Implementation of arbitration policies

The implementation of a round-robin based contention resolution scheme is straightforward, because the bus arbitration process can take place at the penultimate cycle of an outstanding transfer. At that time, the priority is given by examining the pending access requests.

Embedding TDMA or slot-reservation schemes into a bus arbiter is instead a non-trivial task, because in these cases the arbitration process must take place at predefined instants of time, and this might result in a bus preemption for the master that owns it. For single outstanding transfers this is not too much of an issue, because they are so short that they can be considered as atomic and the arbiter has a sufficiently fine-grained control of the bus. The main difficulty lies in the need to support bus preemptions during burst transfers. The master that looses bus ownership in the middle of a burst must be able to properly complete the remaining tranfers once it re-gains access to the bus.

We found that TDMA and slot-reservation based arbitration policies can be implemented in an AMBA arbiter without loosing compliance with the standard by exploiting an option of the AMBA specification called *Early Burst Termination* (EBT). This option was originally meant to support bus preemption so to be able to set an upper bound on the bus ownership time for each master and to prevent other masters from incurring unacceptable access waiting times. Providing support for EBT is a responsibility of the designers of AMBA compliant masters and slaves. The AMBA specification only states that the master that looses bus ownership must re-arbitrate for it in order to complete the burst, and this has to be done by means of any legal burst encoding (e.g. incremental burst of unspecified length).

4.7 Multiprocessor simulation platform

Our experiments have been performed on a complete multiprocessor SoC simulation platform, called MPARM [47]. SystemC [10] is used as backbone simulation engine, and this provides the advantage of describing both hardware and software in a common language (C++).

4.7.1 Hardware support

The efficient integration into the simulation platform of a scalable number of instruction set simulators (ISSs) can be easily carried out by encapsulating them into SystemC wrappers. The ISSs used in our architecture are cycle accurate simulators of cached ARM cores, written in C++ and called SWARM [35].

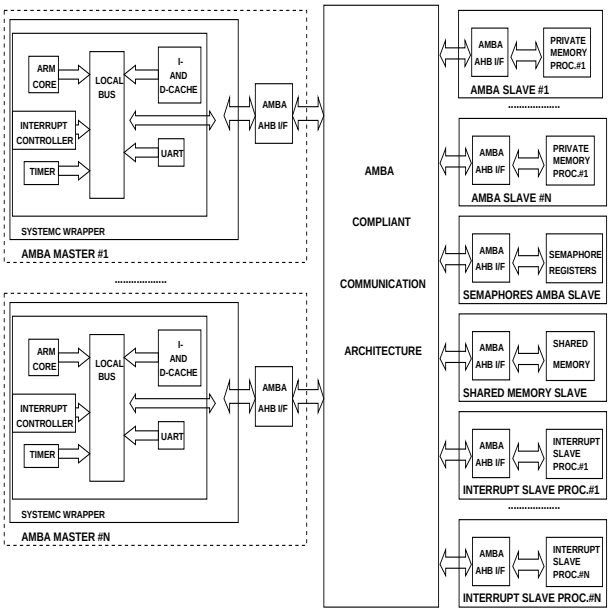


Figure 4.3: Multiprocessor SoC architecture

Along with the CPU, a set of peripherals is emulated (timers, interrupt controller, UART) to provide support for an operating system running on the platform.

The cycle accuracy of SWARM greatly simplifies the synchronization with the SystemC environment, as the control is returned to the SystemC process scheduler at every clock cycle. The simulation platform also includes hardware support for parallel programming. In fact, in this multiprocessor system process synchronization must be ensured, to allow mutually exclusive access of the processes to shared memory resources. To this purpose, we have equipped the simulator with a bank of memory-mapped registers, connected to the AMBA bus as a slave, working as hardware semaphores.

The platform simulates two memory hierarchies: instruction and data caches and main memories. While cache memories are simulated within SWARM, each processing element has its own external private memory which is instantiated as an AMBA slave. Therefore, read or write accesses to the private memory take place through the AMBA system bus and incur contention-related latency. An overview of the system is reported in Fig. 4.3. Up to 32 cores can be instantiated in the system.

Besides private memories, the system includes one shared memory which is used, at the hardware level, only to implement application-level interprocessor communication. For the same purpose, interrupt slaves are instantiated, as outlined in the next subsection.

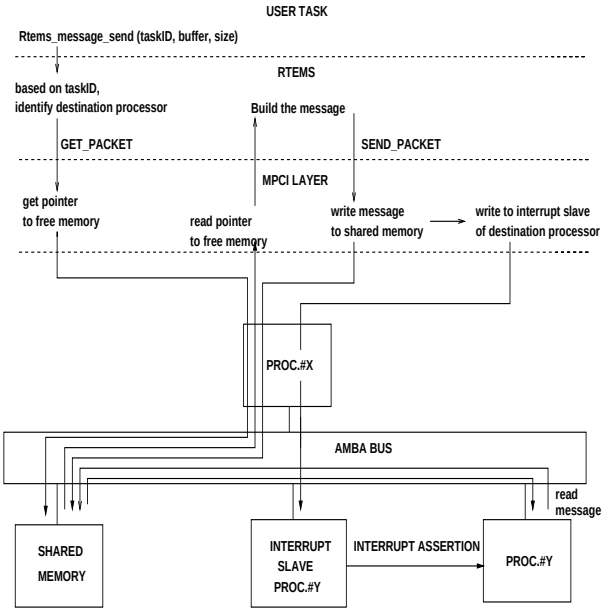


Figure 4.4: Inter-processor communication procedure

4.7.2 Software support

The software support for this simulation platform includes a complete port of an embedded OS, RTEMS [55]. RTEMS is a real-time OS that features POSIX APIs, synchronization and inter-task communication primitives for a multiprocessor scenario.

Inter-processor communication at the application layer takes place through message passing, according to the procedure briefly illustrated in Fig. 4.4. By means of high-level send and receive communication primitives called by a task, the message to be exchanged is read by RTEMS kernel and transferred, in packets, at the MPCI layer. Packets are then written into the shared memory. At this point, we have configured the kernel to use an interrupt based notification technique to signal the destination processor that there is an outstanding packet for it, and we have provided hardware support for this methodology. In particular, we force the source processor to carry out a write transfer to a memory-mapped slave, which asserts a dedicated interrupt of the destination processor. The assertion of this external interrupt triggers a service routine that reads the message from the shared memory, thus completing communication.

It is important to note that inter-processor communication can contribute to a large fraction of bus transactions, increasing the contention for accessing the shared bus. In multiprocessor scenarios wherein synchronization among processors is required for task execution (e.g. exchange of data among processors in distributed signal processing applications), communication-related

traffic could be dominant and the way it is handled by bus arbitration policies could play a key role in determining the overall system performance.

4.8 Performance analysis of arbitration algorithms

Our objective was to stress the distinctive features of the considered arbitration algorithms so to come up with selection guidelines under different system workloads. To this purpose, we identified three scenarios at the application level, corresponding to three different communication patterns: mutually dependent tasks, independent tasks and pipelined tasks.

4.8.1 Mutually dependent tasks

Let us assume a workload wherein one task is running on each processor and that the correct execution of each task involves synchronization with the other ones. In particular, let us assume that all tasks have to synchronize with each other at predefined points of the multiprocessor benchmark. In this case, system performance optimization translates to avoiding that some tasks reach the synchronization point much earlier or much later than the others, because this would generate idle waiting time for the unsynchronized task.

An example thereof is represented by the bootstrap stage of RTEMS on the multiprocessor system. RTEMS selects one processor to act as a master and all other ones are considered as slaves, and they play a slightly different role in the booting operation. Each processor (master and slaves) at first independently initializes its private memory and hardware devices, then synchronization has to take place at the shared memory. In fact, the master processor is in charge of initializing the shared memory and of allocating the structures for inter-processor communication. Then it starts polling the status variables of the slave processors, untill they are all set to ”ACTIVE”, indicating that the slave processors have defined their own data structures in the shared memory. When this synchronization condition occurs, the master processor sets those variables to ”FINISHED”, notifying the slaves that the initialization of the shared memory is over and that each processor can independently complete its bootstrap stage and load its tasks.

We ran the RTEMS bootstrap routine several times, with different arbitration policies implemented in the AMBA arbiter. The performance metric in this scenario is the bootstrap execution time. Each contention resolution scheme is assessed based on its ability to minimize this time and on the associated cost.

Results are shown in Fig. 4.5, where the execution time for a bootstrap on 5 processors is plotted as a function of the slot duration. With slot reservation,

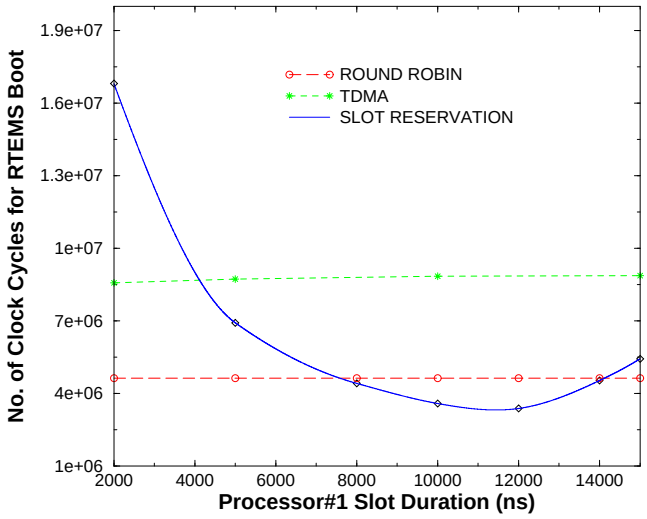


Figure 4.5: Execution time of the bootstrap routine of RTEMS on the multiprocessor platform

the contention-free slot is assigned to the master processor, denoted as *Proc#1*. Its slot duration is reported on the x-axis, while the inter-slot period is kept constant at 1000 cycles. With TDMA, the x-axis refers to the slot allocated to each processor. Finally, with round robin we have no parameters to set, and this corresponds to the constant value observed on the plot.

Round robin exhibits a good performance, depending on the contention level for accessing the bus. Due to asymmetric workload associated to master and slave processors, with a round robin policy we observe that the slaves have to wait for the master (which has more operations to carry out, and in general is more computation-intensive) at the synchronization point, and this slows down the overall RTEMS bootstrap on the multiprocessor system. This suggests to allocate the slot for contention-free access to the bus to the master processor. In fact, for a slot duration of about half the inter-slot period, the slot reservation arbitration policy outperforms round robin, because the increased bandwidth given to the master processor makes up for the asymmetric workload. This minimizes the waiting time of the processors at the synchronization point.

For higher values of the slot duration, too much bandwidth is assigned to the master respect to its needs: the effect is that this time the master reaches the synchronization point much in advance respect to the slaves, and has therefore to wait for them. On the other hand, the small amount of time reserved to slave processors for bus utilization significantly degrades their performance, and the overall execution time increases. An excessively small slot duration

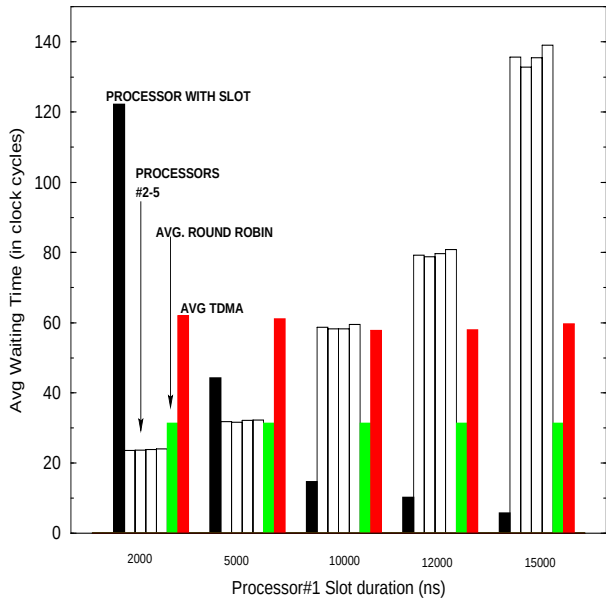


Figure 4.6: Average waiting time of the processors for accessing the bus

causes the same effect, because the master operation slows down respect to a round robin arbitration, and the negative impact on the execution time is even more remarkable because of its asymmetric workload.

Finally, TDMA exhibits the worst performance, in that no balancing effect takes place but only a redistribution of the bus request patterns.

Next, we analyzed the cost incurred by slot reservation for the offered performance. This cost is assessed in terms of average waiting time, defined as the period between the time a processor asserts its bus request signal and the time its grant signal is asserted by the arbiter, indicating that the ownership of the bus has been actually granted. Results are reported in Fig. 4.6. The average waiting time of the master processor is compared to that of the other processors when slot reservation is activated. With the other policies, as the average waiting times of all processors are more balanced, only the overall average value is reported.

It is interesting to observe that for an optimal slot duration of about 12000 ns (600 clock cycles, the clock period being 20 ns) derived from the previous plot, the average waiting time of the high priority master is more than halved and that of the other processors is more than doubled respect to the round robin case. This effect does not play any role in this context, as the performance metric of interest is the minimization of the total execution time. Therefore, an increase of the latency for accessing the bus can be sometimes tolerated, provided it is not directly related to the system performance perceived at the application level.

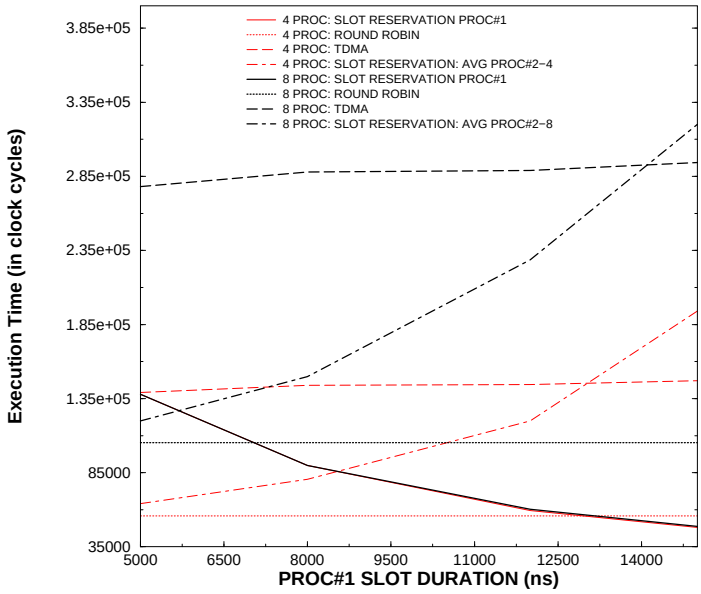


Figure 4.7: Execution time for a benchmark made by independent task

4.8.2 Independent tasks

The second scenario we investigated makes use of a benchmark consisting of independent tasks, each running on a specific processor. This system workload does not have any synchronization point, nor it involves inter-processor communication.

The above scenario has been implemented on our simulation platform by executing the same matrix multiplications on each processing element. Matrixes are initially stored in each processor's private memory, and the traffic generated on the bus is associated with read operations of matrix elements and to write transactions storing the results back in the memory. Tasks execution and consequent measurements are triggered once RTEMS has booted on all of the processors.

The performance metric we select for this class of benchmarks is the average task execution time, given the independent nature of the tasks themselves. Our experiments have been carried out ranging the number of processors from 2 to 10, analyzing the scaling properties of the performance metric.

Results relative to the tasks execution times are reported in Fig. 4.7, for the cases of 4 and 8 active processors. When 4 tasks are running, we observe that round robin outperforms the other schemes. In fact, if we randomly select one processor (e.g. processor no.1) and periodically grant it a slot for contention-free access to the bus, the improvement of its execution time translates to a relevant degradation of the performance for the other processors, and the av-

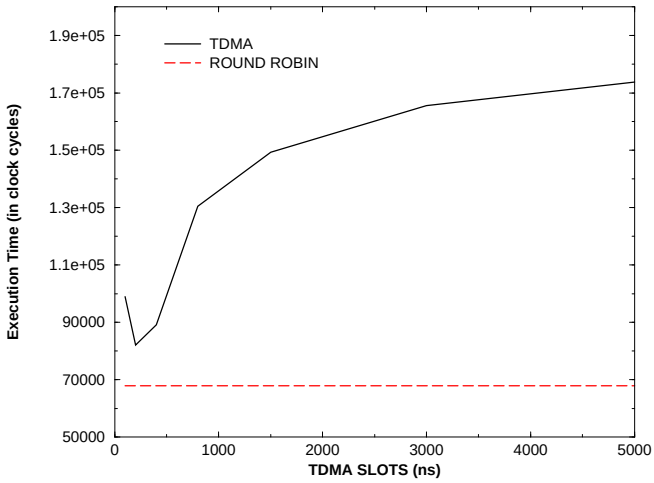


Figure 4.8: Comparison between performance of round robin and TDMA for small values of TDMA slot

erage task execution time of the system increases. Though it is interesting to observe that a slot allocation of 9000 ns manages to balance the execution times of all processors, so that on average all tasks complete within the same time, similarly to what happens with round robin or TDMA, and this is the most efficient approach for this scenario. The relevant difference between the three arbitration algorithms is in the average execution time that can be obtained by each of them under the hypothesis of balanced task execution times. The balancing effect for slot reservation (achieved by properly tuning the slot duration) occurs at an average execution time which lies between that provided by round robin (the optimal one) and that provided by TDMA (worst case).

The same effect can be observed with 8 processors, even though the average values increase and the gap between round robin and slot reservation decreases.

One might guess that the performance of TDMA is likely to increase for smaller values of TDMA slot respect to those reported in Fig 4.7, so to reduce bus idleness. The answer to this question is reported in Fig. 4.8, where the average execution time of the tasks is plotted as a function of (smaller) TDMA slot. Although the performance offered by TDMA actually increases, it never performs like round robin. The shortest execution time occurs when the TDMA slot is in the order of the duration of a burst transfer. For smaller values, bus preemptions start playing a dominant role and their high frequencies (and their associated costs in terms of bus cycles) determine a performance degradation. The behaviour showed in Fig. 4.8 can be explained with the redistribution of request patterns operated by TDMA and by the misalignment of such patterns

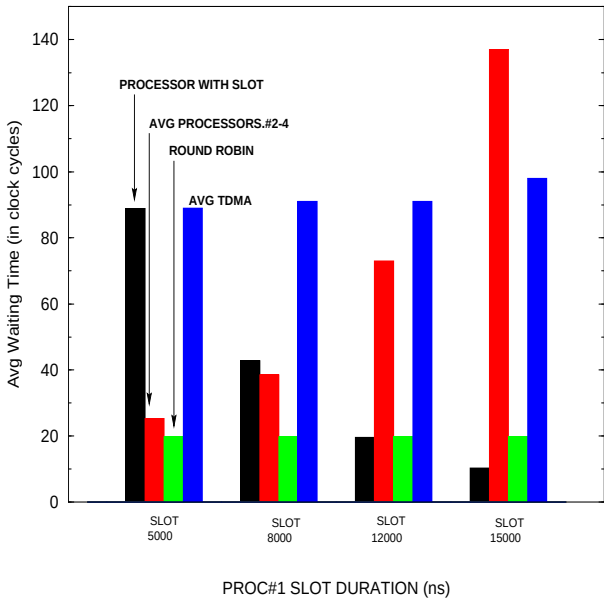


Figure 4.9: Bus access delays for the benchmark with independent task

with the slot allocation.

Going back to Fig. 4.7, another consideration is worth mentioning. Let us assume that one processor has to be allocated more bandwidth, tolerating the performance degradation of other schemes. For this processor, a slot must be allocated such that its final execution time be less than that exhibited in the round robin case. Therefore, if we check the crossing point between the "Proc#1" curve and the round robin curve, we see that as the number of processors increases, the slot duration that ensures such a performance decreases (it is almost halved from 13000 ns to 7000 ns), and this is tightly related to the increased contention levels on the bus.

As regards the average waiting time for accessing the bus, an histogram is reported in Fig. 4.9. For a slot value of 8000 ns, close to the optimal value that balances the execution times, the average waiting times of the processors are balanced as well, but higher than that achievable by means of a round robin arbitration. Note the very poor performance of TDMA for such values of the slot.

Finally, we want to show how the execution times of the processors scale as a function of the number of active processors. This result is reported in Fig. 4.10: the values for the high priority processor (e.g. "slotx-I", where x is the slot duration) are compared with the average ones of the remaining processors ("slotx-avg") when slot reservation is used, and with the average round robin and TDMA values ("tdmax"). It is interesting to observe the rapidly degrading performance of TDMA, while round robin and slot reservation are

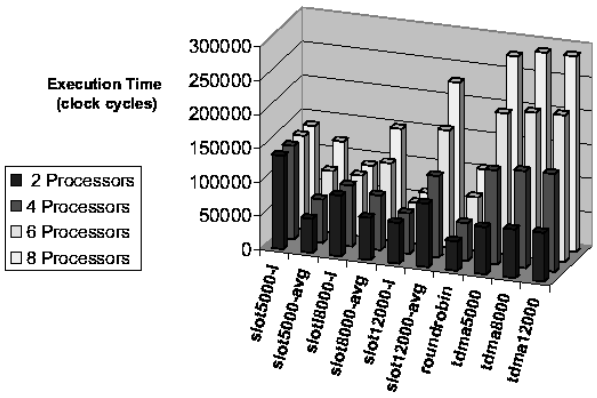


Figure 4.10: Scalability property of the execution times with different arbitration policies

more scalable under this point of view.

4.8.3 Pipelined tasks

A last scenario that is worth investigating is the one wherein the impact of arbitration policies on the throughput of a distributed signal processing application can be assessed. While in the previous subsection we analyzed a system workload wherein the traffic across the bus did not depend on inter-processor communication at all, but was only related to computation (e.g. cache line refills), now we want most of bus transactions to be related to communication among processors. We want to relate the performance of such a system to the way communication related traffic is accommodated on the bus by the different arbitration policies.

To this purpose, we set up a multiprocessor system wherein different tasks execute in a pipelined fashion, with balanced computation workloads for all of the processors (they execute matrix multiplications). On top of the first processor, a task generates matrixes that are handed over to the second processor of the pipeline. At each stage, the computation is carried out and the result transmitted to the next stage. In other words, the pipeline consists of couples of producer-consumer tasks, and the communication occurs, at a high level of abstraction, by means of FIFO queues.

The performance metric for this system is the throughput, defined as the number of matrixes per second produced by the last processor of the pipeline (i.e. frame rate).

Fig. 4.11 shows the frame rate provided by the arbitration policies, chang-

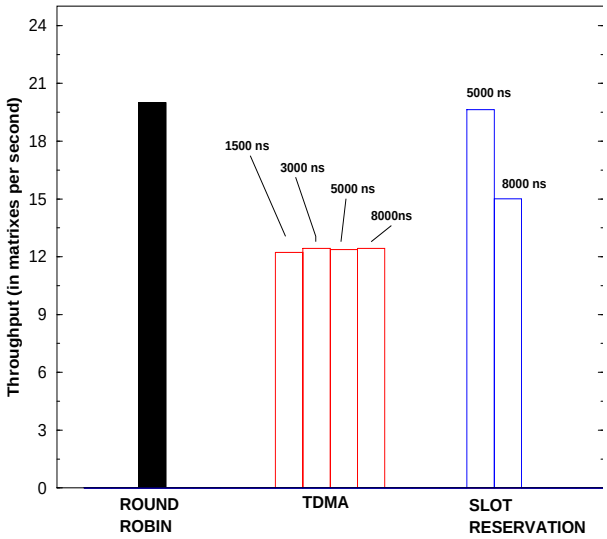


Figure 4.11: Throughput of the system for different arbitration schemes.

ing the value of the slot duration for slot reservation and TDMA. While the performance of slot reservation is highly sensitive to the slot time, the performance of TDMA is almost independent of it. Surprisingly enough, although both the workload and the communication needs of the pipelined processors are perfectly balanced, slot reservation performs better than TDMA for a wide range of slot durations. This can be explained by looking at the performance of round robin, that is always much better than TDMA. Since our slot reservation algorithm implements a round robin arbitration policy during inter-slot times, as long as the slot duration is much shorter than the inter slot time, the performance of slot reservation is dominated by the performance of round robin, while it becomes much worse when larger slots are used. Therefore, in Fig. 4.11 only two experiments for slot reservation have been carried out, because they are sufficient to clarify the dependence of execution time as a function of the slot duration.

Since the frame rate provided by slot reservation is always smaller than that of round robin, we can say that slot reservation is counter productive in this case. In fact, there is no reason for guaranteeing a constant bandwidth to a single stage of a pipeline if the same bandwidth cannot be guaranteed to all stages.

On the other hand, TDMA guarantees a constant bandwidth to all processors in the pipe, but its overall performance is lower than that of round robin. This fact can be explained only by looking at the hardware implementation of high level inter-processor communication primitives.

In our system, the producer-consumer paradigm is implemented by means of the RTEMS message manager, which makes use of a communication proto-

col among tasks based on message queues. At the core of this protocol there is the inter-processor communication mechanism seen in Fig. 4.4. The procedure is initiated by the producer, which creates a global queue in its private memory, and writes messages to be sent in it.

When the consumer is ready to receive a message, a notification is given to the producer by writing a request message into the shared memory and by generating an interrupt for the producer itself. The interrupt service routine of the producer reads the message from shared memory and assembles data to be sent in a message which is written back to shared memory. Finally, a write transaction to the consumer interrupt slave asserts an interrupt which allows the consumer to pick up its message from shared memory.

In thix context, TDMA poor performance can be explained in terms of its inability to support the communication handshake between the producer and the consumer, which is necessary for the hardware implementation of the high level inter-processor message passing. This handshake involves a ping-pong interaction between the two tasks, and is inefficiently accommodated in a TDMA based architecture, wherein only one processor is active during each slot. This results in a higher latency for the interaction respect to the round robin case, and this explains the poor performance of TDMA observed in the experiments.

This low level implementation of message passing primitives made available by RTEMS to the applications involves a large overhead in terms of bus transactions. This overhead may result in a relevant system performance penalty, and derives from a mismatch between the software architecture and the underlying hardware platform. In other words, these two layers should be aware of each other to maximize system performance.

As an example, it is worth mentioning that our multiprocessor simulation platform does not support global cache coherency, therefore the shared memory is declared ”non-cacheable”. Furthermore, the ARM ISS embedded into our platform does not support burst transfers except for cache line refills (accessing only private memories). As a consequence, reading or writing data to shared memory is a highly inefficient operation, because it only takes place by means of single transfers instead of burst transfers.

Finally, we observe that the poor performance exhibited by TDMA is also related to the fact that it is inefficiently accommodated in an AMBA based communication architecture. In fact, the ultimate objective of the AMBA bus protocol is contention avoidance, and the signals used by masters and slaves have to be seen under this perspective (e.g. HBUSREQ, etc..). On the contrary, TDMA would require a simpler communication protocol, as the whole contention management procedure is arbiter driven. As a consequence, TDMA

might outperform other arbitration algorithms in proprietary communication architectures.

Despite the lower performance, TDMA-based arbitration is also attractive in many real-time applications where predictability is a critical requirement. In fact, TDMA reserves a slot to each processor regardless of the current workload, thus making constant in time the bandwidth perceived by each processor, independently of the traffic generated by the other masters. Consider, for instance, a system composed of 10 processor cores. If 5 of the cores are used to implement the pipelined streaming application described in this subsection the frame rate achieved will be constant and predictable, independently of the traffic generated by the processors that do not take part in the pipeline (hereafter called external processors). Using round robin, the frame rate would be much better than that provided by TDMA when the traffic generated by the external processors is negligible, but it would be strongly dependent on the overall workload, possibly becoming worse than that of TDMA when external processors perform memory/communication intensive tasks. Non-determinism is not acceptable in many real-time situations.

4.9 Conclusions

In this work we analyze the impact on multiprocessor SoC performance of different bus arbitration policies under different communication patterns, showing the distinctive features of each policy and the strong correlation of their effectiveness with the communication requirements of an application.

Beyond two traditional bus arbitration policies (round robin and TDMA) we consider another technique that periodically allocates fixed predictable bandwidth to time-critical processors ("slot reservation"). Three workloads are analyzed on our multiprocessor simulation platform (mutually dependent tasks, independent tasks and pipelined tasks), and some important guidelines for designers of SoC communication architectures have been derived:
(i) the optimal bus arbitration policy is not unique, but strongly depends on the traffic conditions (computation-dependent, communication-dependent,etc..).
(ii) The software support for inter-processor communication plays a crucial role in determining system performance, as it has to be matched with the underlying hardware platform. High level communication primitives, although facilitating the programming step, could be inefficiently implemented on the available platform, degrading system performance.
(iii) There exists a trade-off between contention-resolution bus arbitration policies (such as TDMA) and contention-avoidance bus protocols (such as AMBA bus). Even though commercial standards provide degrees of freedom for per-

formance optimization, the performance achievable by contention-resolution policies implemented within contention- avoidance protocols cannot be fully exploited, because of the different characteristics of these two elements.

Chapter 5

Exploring Programming Models and their Architectural Support

5.1 Abstract

In today's multi-processor SoCs (MPSoCs), parallel programming models are needed to fully exploit hardware capabilities, and to achieve the 100 Gops/W energy efficiency target required for Ambient Intelligence Applications. However, mapping abstract programming models onto tightly power-constrained hardware architectures imposes overheads which might seriously compromise performance and energy efficiency.

The objective of this work is to perform a comparative analysis of message passing versus shared memory as programming models for single-chip multiprocessor platforms. Our analysis is carried out from a hardware-software viewpoint: we carefully tune hardware architectures and software libraries for each programming model. We analyze representative application kernels from the multimedia domain, and identify application-level parameters that heavily influence performance and energy efficiency. Then, we formulate guidelines for the selection of the most appropriate programming model and its architectural support.

5.2 Introduction

The traditional dichotomy between shared memory and message passing as programming models for multi-processor systems has consolidated into a well-

accepted partitioning. For small-to-medium scale multi-processor systems there is an undisputed consensus on cache-coherent architectures based on shared memory. In contrast, large-scale high-performance multi-processor systems have converged towards non-uniform memory access (NUMA) architectures based on message passing (MP) [60, 61].

The appearance of Multi-Processor Systems-on-Chip (MPSoCs) in the multi-processing scenario, however, has somehow put this picture in discussion. Several peculiarities differentiate in fact these architectures from classical multiprocessing platforms. First, their "on-chip" nature reduces the cost of inter-processor communication. The cost of sending a message on an on-chip bus is in fact at least one order of magnitude lower (power- and performance-wise) than that of an off-chip bus, thus pushing towards message passing-based programming models. On the other hand, the cost of on-chip memory accesses is also smaller with respect to off-chip memories; this makes cache-coherent architectures based on shared memory competitive.

Second, MPSoCs are resource-constrained systems. This implies that while performance is still critical, other cost metrics such as power consumption must be considered. Unfortunately, it is not usually possible to optimize power and performance concurrently, and one quantity must typically be traded off against the other one.

Third, unlike traditional message passing systems, some MPSoC architectures are highly heterogeneous. For instance, some platforms are a mix of standard processor cores and application-specific processors such as DSPs or micro-controller [76, 65]. Conversely, other platforms are highly modular and reminiscent of traditional multi-processor architectures [79, 81]. While in the former case message-passing is the only viable alternative (some of the processing engines may even be cacheless), in the latter case a cache-coherence model seems to be the most intuitive choice.

All these issues indicate that the choice between the two programming models is not so well-defined for MPSoCs. The objective of this work is precisely that of exploring what factors may affect this choice, yet from a novel and more exhaustive perspective. Although our analysis considers the two traditional dimensions of the problem, namely, the *architecture* and the *software*, they are both considered from the software perspective. In particular, we assume that the variable '"architecture" is determined by the programming model. The actual dimension becomes then the *programming model* (shared-memory vs. message-passing), under the assumption that *to each model corresponds an underlying architecture that is optimized for it*.

This assumption, which is at the core of this work, stems from considering the inefficiency incurred when mapping high-level programming models (such

as message passing) onto generic architectures, in terms of software and communication overhead. This conflicts with the trend of designing optimized, custom-tailored architectures showing very high power and communication efficiency in a restricted target application domain (application-specific MPSoCs).

On the software side, conversely, we consider more traditional parameters, the most important being *the workload allocation strategy*. However, we also consider more application-specific parameters that affect the communication (e.g., the size of the messages or the communication/computation ratio).

Unlike previous works, we do not simply do a re-writing of benchmarks under different programming models for a given architecture. In our case, using a different model implies using a different architecture, and the software is modified accordingly so as to exploit the optimized communication features provided by the hardware. It is worth emphasizing that we do not want to demonstrate the superiority of one paradigm over the other. Rather, we show that, for a given target application, there may not be a programming model which is consistently better than the other. Our focus is on media and signal processing applications commonly found in MPSoC platforms.

Our exploration leverages an accurate multi-processor simulation environment that provides cycle-accurate simulation and estimation of power consumption, based on $0.13\mu m$ technology-homogeneous industrial power models, see [97].

In summary, the main contributions of our work are: (i) the creation of a flexible and accurate MPSoC performance and power analysis environment; (ii) the development of highly optimized hardware assists and software libraries for supporting message passing and shared memory programming abstractions on an MPSoC platform; (iii) comparative energy and performance analysis of message passing and shared memory hardware and software tuned MPSoC architectures for coarse-grain parallel workloads typical of the multimedia application domain; (iv) derivation of general guidelines for matching a task-level parallel application with a target hardware-software platform.

5.3 Related Work

Parallel programming and parallel architectures have been extensively studied in the past forty years in the domain of high-performance general-purpose computing [60]. Our review of related works focuses primarily on multiprocessor SoC architectures for embedded applications [80, 76, 77, 78, 74].

From the software view-point, there is little consensus on the programmer view offered in support of these highly parallel MPSoC platforms. In many

cases, very little support is offered and the programmer is in charge of explicitly managing data transfers and synchronization. Clearly, this approach is extremely labor-intensive, error-prone and leads to poorly portable software. For this reason MPSoC platform vendors are devoting an increasing amount of effort to offering more abstract programmer views through middleware libraries and their APIs. *Message passing* and *shared memory* are the two most common approaches.

Message passing has first been studied in the high-performance multiprocessor community, where many techniques have been developed for reducing message delivery latency[67, 68, 66]. Message passing has also entered the world of embedded MPSoC platforms. In this context it is usually implemented on top of a shared memory architecture (e.g. TI OMAP[78], Philips Eclipse [65], Toshiba Kawasaki[64], Philips Nexperia[76]). Hence, shared memory is likely to become a performance/energy bottleneck, even when DMAs are used to increase the transfer efficiency.

Therefore, several authors have recently proposed support for message-passing on a distributed memory architecture. Two interesting case studies are presented in [63, 62] The above approaches have limited support for synchronization and limited flexibility in matching the application to the communication architecture. E.g., in [62] remote memories are always accessed with a DMA-like engine even though this is not the most efficient strategy for small message sizes.

Even though message passing has received some attention, shared memory is the most common programmer abstraction in today's MPSoCs. However, the presence of a memory hierarchy with locally cached data is a major source of complexity in shared-memory approaches. Widely speaking, approaches for solving the cache coherence problem fall into two major classes: hardware-based approaches, and software-based ones. The former imposes cache coherence by adding suitable hardware which guarantees coherence of cached data [103, 104, 60], whereas the latter imposes coherence by limiting the caching of shared data [105]. This can be done by the programmer, the compiler, or the operating system.

In embedded MPSoC platforms, shared memory coherence is often supported only through software libraries which rely on the definition of non-cacheable memory regions for shared data or on cache flushing at selected points of the execution flow. However, there are a few exceptions that rely on hardware cache coherence, especially for platforms which have a high degree of homogeneity in computational node architecture [81].

The literature on comparing message passing and shared memory as programming models in large-scale general-purpose multiprocessors is quite rich

([82]–[89]). Early works ([82, 83, 84]) compare a shared memory program against a similar program written with a message passing library that was implemented in shared memory on the same machine. The first two works provide strong evidence of the superiority of message passing, a conclusion which the third work partially puts in discussion.

These works do not actually explore programming styles, since they do not use the architectural variable. The performance of a message passing library simulated on a shared memory computer is likely to be quite different from the more complex library on message passing hardware. Also, the programs were executed on a real machine, which limited the comparison to elapsed time.

Simulation was used in [85] to compare message traffic in the two programming models, by writing applications in a parallel language that supports high-level communication primitives of the two types. Translation onto the target architecture is done through a compiler, which however affects the interpretation of the comparison. Chandra et al. [86] did a more predictable analysis by careful writing of the application onto the same hardware platform. Their conclusions partially upset the superiority of message passing in favor of a shared memory paradigm. More recent works ([87, 88, 89]) focused again on specific platforms such as high-end SMPs.

From our perspective, these works have several limitations, which we address in our analysis. First, and foremost, all methods but [86] refer to a specific architecture, which is thus not considered as a dimension of the exploration. Second, none of them explicitly refers to MPSoCs as an architectural target, therefore power or energy are never considered as valuable design metrics. Third, non-realistic software architectures are sometimes considered (e.g., [108, 109]).

5.4 Hardware Architectures

The architecture of the hardware platform is designed to provide efficient support for the different styles of parallel programming. Therefore, our MPSoC simulation platform was extended in order to model and simulate the following architectures:

5.4.1 Shared memory architecture

This architecture consists of a variable number of processor cores (ARM7 simulation models will be deployed for our analysis framework) and of a shared memory device to which the shared addressing space is mapped.

As an extension, each processor also has a private memory connected to the

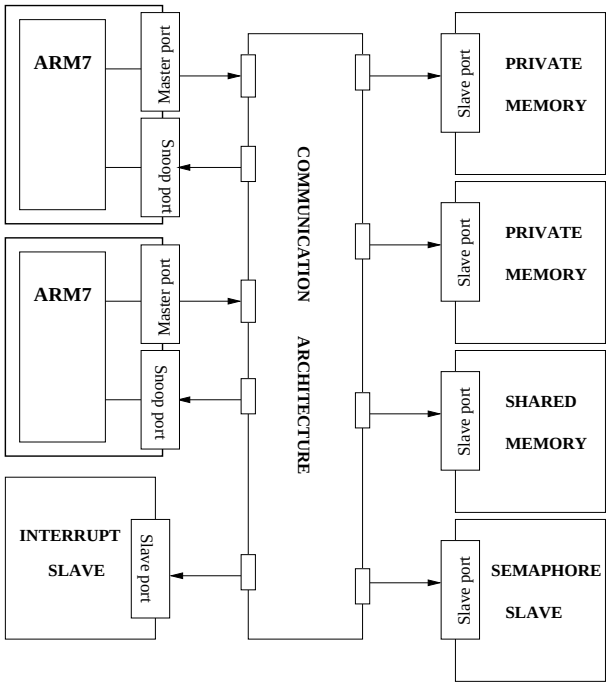
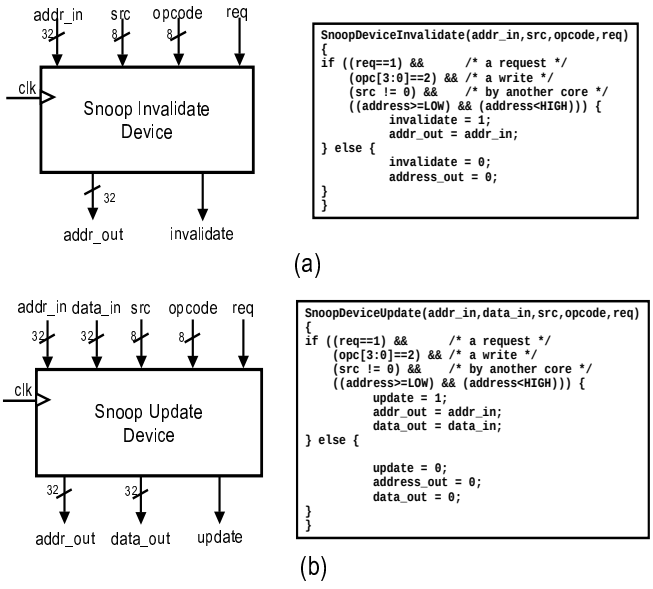


Figure 5.1: Shared memory architecture.




```
SnoopDeviceInvalidate(addr_in,src,opcode,req)
{
if ((req==1) &&      /* a request */
    (opc[3:0]==2) && /* a write */
    (src != 0) &&    /* by another core */
    ((address>=LOW) && (address<HIGH))) {
          invalidate = 1;
          addr_out = addr_in;
} else {
          invalidate = 0;
          address_out = 0;
}
}
```

```
SnoopDeviceUpdate(addr_in,data_in,src,opcode,req)
{
if ((req==1) &&      /* a request */
    (opc[3:0]==2) && /* a write */
    (src != 0) &&    /* by another core */
    ((address>=LOW) && (address<HIGH))) {
          update = 1;
          addr_out = addr_in;
          data_out = data_in;
} else {

          update = 0;
          address_out = 0;
          data_out = 0;
}
}
```

Figure 5.2: Interface and Operations of the *Snoop Device* for the Invalidate (a) and Update (b) Policies.

bus where it can store its own local variables and data structures (see Fig. 5.1). In order to guarantee data coherence from concurrent multiprocessor accesses, shared memory can be configured to be non-cacheable, but in this case it can

only be inefficiently accessed by means of single bus transfers.

This inefficiency might be overcome by creating copies of shared memory locations in private memory (i.e., using shared memory only as a communication channel). Data would then become cacheable and could be accessed via burst transfers at the cost of moving a larger volume of data through the bus.

Alternatively, the shared memory can be declared cacheable, but in this case cache coherence has to be ensured. We have enhanced the platform by adding a hardware coherence support based on a *write-through policy*, which can be configured either as *Write-Through Invalidate, WTI* or *Write-Through Update, WTU*.

The hardware snoop devices, for both invalidate and update case, are depicted in Figure 5.2. The snoop devices sample the bus signals to detect the transaction which is being performed on the bus, the involved data and the originating core. The input pinout of the snoop device depends of course on the particular bus implemented in the system, and Figure 5.2 reports the specific example of the interface with the STBus interconnect from STMicroelectronics, although signal lines with identical content can be found in most communication architecture specifications.

When a write operation is flagged, the corresponding action is performed, i.e., invalidation for the WTI policy, rewriting of the data for the WTU one. Write operations are performed in two steps. The first one is performed by the core, which drives the proper signals on the bus, while the second one is performed by the target memory, which sends its acknowledge back to the master core to notify operation completion (there can be an explicit and independent response phase in the communication protocol or a ready signal assertion in a unified bus communication phase). The write ends only when the second step is completed and when the snoop device is allowed to consistently interact with the local cache. Of course, the snoop device must ignore write operations performed by its associated processor core. In our simulation model, synchronization between the core and the snoop device in a computation tile is handled by means of a local hardware semaphore for mutually exclusive access to the cache memory.

Hardware semaphores and slaves for interrupt generation are also connected to the bus (Fig. 5.1). The interrupt device allows processors to send interrupt signals to each other. This hardware primitive is needed for interprocessor communication and is mapped in the global adressing space. For an interrupt to be generated, a write should be issued to a proper address of the device. The semaphore device is also needed for the synchronization among the processors; it implements test-and-set operations, the basic requirement to have semaphores.

Further details of the shared memory architecture can be found in table 5.1.

processor	ARM7	200Mhz	5 pipeline stage
data cache	up to 4KByte	4 way set associative	latency 1 cycle
instruction cache	4KByte	direct mapped	latency 1 cycle
scratchpad	up to 8KByte	200Mhz	latency 1 cycle
private memory	128KByte	200Mhz	latency 2 cycle
shared memory	256KByte	200Mhz	latency 2 cycle
STBUS	32 bit	200Mhz	split bus

Table 5.1: Technical details of the architectural components

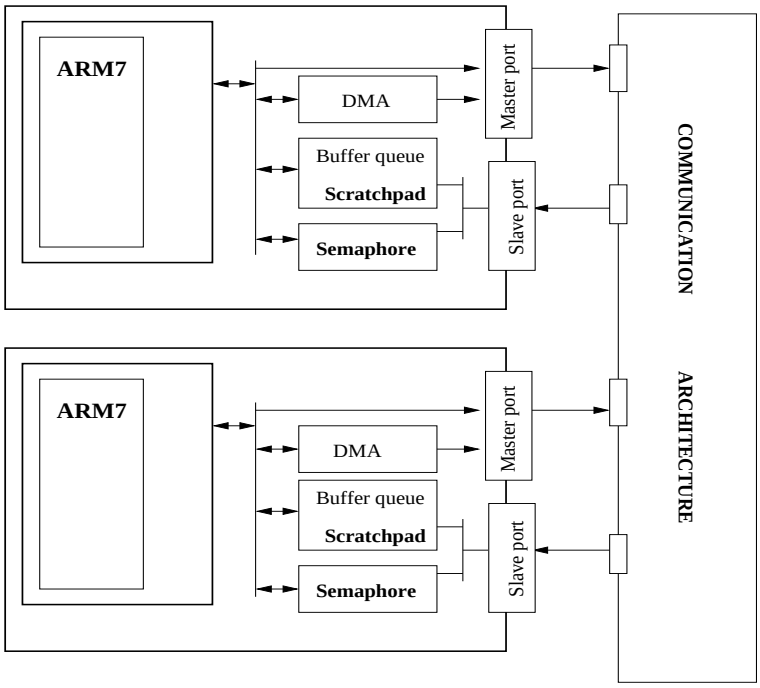


Figure 5.3: Message-oriented distributed memory architecture.

The template followed by this shared memory architecture reflects the design approach of many semiconductor companies to the implementation of shared memory multi-processor architectures. As an example, the MPCore processor implements the ARM11 micro-architecture and can be configured to contain between 1 to 4 processor cores, while supporting fully coherent data caches[75].

5.4.2 Message-oriented distributed memory architecture

Message passing helps mastering the design complexity of highly parallel systems provided the transfer cost on the underlying architecture can be limited. We therefore consider a distributed memory architecture with light-weight hardware extensions for message passing, as depicted in Fig. 5.3.

In the proposed architecture, a scratchpad memory, a semaphore and DMA unit are attached to each processor core. The different processor tiles are con-

nected using the shared bus (STBus). In order to send a message, a producer writes in the message queue stored in its local scratch-pad memory, without generating any traffic on the interconnect. Once the data is in the message queue, the corresponding consumer (running on another processor) can fetch the message to its own scratch-pad, directly or via a DMA controller. For this purpose, the scratchpad memories are connected as slaves to the communication fabrics and their space is made visible to any other processor on the platform. The DMA engine attached to each core enables efficient data transfers between scratch-pad and non-local memories (cfr. [100]): it supports multiple outstanding data channels and has a dedicated connection for fast access to the local scratch pad memory.

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 incrementing a semaphore located in its scratch-pad memory. 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.

The details of the message passing architecture can be found in table5.1.

The architecture of the recently announced Cell Processor[74] developed by Sony, IBM and Toshiba shares many similarities with the template we are considering in this paper. The Cell processor exhibits eight vector computers equipped with local storage and connected through a data-ring based system interconnect. The individual processing elements can use this bus to communicate with each other, and this includes the transfer of data in between the units acting as peers of the network.

5.5 Software support

A software library is an essential part of any today's multi-processor system. In order to support software developers in programming the two optimized hardware platforms, we have implemented two architecture-specific communication and synchronization libraries exposing high level APIs. The ultimate objective is to abstract low level architectural details to the programmers, such as memory maps, management of hardware semaphores and intermediate data transfers, while keeping the overhead introduced by the programming library as low as possible, from a performance and power viewpoint.

Concerning the shared memory architecture, we opted for porting a standard communication library onto the MPSoC platform: it is the SystemV IPC Library, which is the native communication library for heavy-weight processes under the Unix operating system. This allows software designers to develop their applications on host PCs and to easily port their code onto the MPSoC virtual platform for validation and fine-grained software tuning on the target architecture.

As regards the message-oriented architecture, it is rather tuned for MPSoC implementations, and its effectiveness was proved in [101] As a consequence, we needed a communication library able to fully exploit the features of this architecture. Moreover, we expect that the porting of standard message passing libraries traditionally used in the parallel computing domain might cause an overly significant overhead in resource-constrained MPSoCs. For this reason, we had to develop our own optimized message passing library, custom-tailored for the scratch-pad-based distributed memory architecture we are considering.

5.5.1 A light-weight porting of System V IPC library for shared memory programming

Brief introduction to IPC standard

System V IPC is a communication library for heavy-weight processes based on permanent kernel resident objects. Each object is identified by a unique kernel ID. These objects can be created, accessed and manipulated only by the kernel itself, granting mutual exclusion between processes. Three different types of objects, named facilities, are defined: messages queues, semaphores and shared memory. Processes can communicate through System V IPC objects using ad-hoc defined APIs, that are specific for each facility.

Message Queues are objects similar to pipes and FIFOs. A message queue allows different processes to exchange data with each other in the form of messages in compliance with the FIFO semantic. Messages can have different sizes and different priorities. The send API (*msgsnd*) puts a message in the queue, suspending the calling process if there is no enough free space. On the other hand, the receive API (*msgrcv*) extracts from the queue the first message that satisfies the calling process requests in terms of size and priority. If there is not a valid message or if there are no messages at all the calling process is suspended until a valid message is written to the queue. A special control API (*msgctl*) allows processes to manage and delete the queue object.

Semaphore objects consist of a set of classic Dijkstra's semaphores. A process calling the "operation" API (semop) can wait and signal on any semaphore

of the set. Moreover, System V IPC allows processes to request more than one operation on the semaphore set at the same time. That API ensures that the operations will be executed atomically. A special control API (*semctl*) allows to initialize and delete the semaphore object.

Shared memory objects are buffers of memory which a process can link to its own memory space through the attach API (*shmat*). All processes which have attached a shared memory buffer see the same buffer and can share data directly reading and writing on it. As the memory spaces of the processes are different, the shared buffer could be attached by the attach API at different addresses for each process. Therefore, processes are not allowed to exchange pointers which refer to the shared buffer. In order to successfully share a pointer, its absolute address must be changed into an offset relative to the starting location of the shared buffer. A special control API (*shmctl*) allows processes to mark a buffer for destruction. A buffer marked for destruction is removed from the kernel when there are no more processes that are linked to it. A process can unlink a shared buffer from its memory space using the detach API (*shmdt*).

Implementation and Optimizations

Some implementation details concerning the MPSoC communication library compliant with the System V IPC standard follow. All the objects, which require to be accessed in a mutually exclusive way, are stored in the shared memory. Therefore, a dynamic allocator was introduced in order to efficiently implement data allocation in shared memory. All original IPC kernel structures were optimized by removing many process/permission related information, in order to reduce shared memory occupancy and therefore API overhead. In our library implementation targeting MPSoCs, mutual exclusion on the critical sections of an object was ensured by means of hardware mutexes that are accessible on the shared memory space. Each IPC object is protected by a different hardware mutex, allowing parallel execution on different objects.

MPSoC platforms are typically resource-constrained. Therefore, we decided not to implement some of the features of System V IPC. At the moment, the priority in the message queues facility and the atomic multi-operations on the semaphore sets have not been implemented. These features are not critical in System V IPC, so that their lack will only marginally affect code portability.

MPSoC IPC library was tested and optimized to improve performance of APIs. The length of the critical sections was reduced as much as possible in order to optimize code efficiency. Similarly, the number of shared memory accesses was significantly reduced. Moreover, in case of repeated read accesses to the same memory location, we hold the read value. Write operations were

optimized avoiding to perform useless write accesses to shared memory (e.g., writing the same value).

Since the benchmarks we will use in the experimental results make extensive use of the semaphore facility, we assessed the cost inucurred by our library in managing this facility. We created an ad-hoc benchmark where two tasks are running onto two different processors: the first one periodically releases a certain semaphore, while the second one is waiting on that semaphore. We measured the time to perform signal and wait over 40 iterations. It turned out that the overhead for using System V IPC with respect to the manual management of the hardware semaphores is negligible (only 2%).

Dynamic memory allocation will never be exploited by our benchmarks since they allocate shared memory during initialization and free it before exiting, therefore we excluded those two phases from system performance measurements. Moreover, we do not use message queues, which involve mapping a message passing paradigm on top of shared memory, i.e. on top of an architecture which is not optimized for messaging, and this goes in the opposite direction with respect to our initial assumptions.

5.5.2 Message Passing library

We also built a set of high-level APIs to support a message passing programming style on the message-oriented distributed memory architecture described above. Our library simplifies the programming stage and is flexible enough to explore the design space. The most important functions are listed in Table 5.2.

Return Type	Function	Arguments
SQ_PRODUCER*	*sq_init_producer*	int consumer_id int message_size int total_messages bool use_suspension
SQ_CONSUMER*	*sq_init_consumer*	int consumer_id bool buffer_space_location bool use_suspension
void	*sq_write(_dma)*	SQ_PRODUCER *queue_p char *source
char*	*sq_getToken_write*	SQ_PRODUCER *queue_p
void	*sq_putToken_write*	SQ_PRODUCER *queue_p
char*	*sq_read(_dma)*	SQ_CONSUMER *queue_c

Table 5.2: APIs of our message passing library

To instantiate a queue, both the producer and consumer must run an initialization routine. To initialize the producer side, the corresponding task must call *sq_init_producer*. It takes as arguments the identifier of the consumer, the message size, the number of messages in the queue and a binary value. The last argument specifies whether the producer should poll the producer's semaphore

or suspend itself until an interrupt is generated by the semaphore. The consumer is initialized with *sq_init_consumer*. It requires the identifier of the consumer itself, the location of the read buffer and the poll/suspend flag. In detail, the second parameter indicates the address where the function sq_read will store the message transferred from the producer's message queue. This address can be mapped either to the private memory or to the local scratch-pad memory.

The producer sends a message with the *sq_write(_dma)* function. This function copies the data from **source* to a free message block inside the queue buffer. This transfer can either be carried out by the core or via a DMA transfer (*x_dma*). Instead of copying the data from **source* into a message block, the producer can decide to directly generate data in a free message block. The *sq_getToken_write* returns a free block in the queue's buffer on which the producer can operate. When data is ready, the producer should notify its availability to the consumer with *sq_putToken_write*. The consumer transfers a message from the producer's queue to a private message buffer with *void sq_read(_dma)*. Again, the transfer can be performed either by a local DMA or by the core itself.

Our approach thus supports: (1) either processor or DMA-initiated data transfers to remote memories, (2) either polling-based or interrupt-based synchronization, and (3) flexible allocation of the consumer's message buffer, i.e. on scratch-pad or on a private memory at a higher level of the hierarchy.

Low overhead implementation and tuneability

The library implementation is very light-weight, since it is based on C macros that do not introduce significant overhead with respect to the manual management of hardware resources. A producer-consumer exchange of data programmed via the library showed just a 1% overhead with respect to a manual control of the transfer by the programmer without high level abstractions.

More interestingly, the library flexibility can be used for fine tuning the porting of an application on the target architecture. In fact, the library can exploit several features of the underlying hardware such as processor- versus DMA-driven data transfers or interrupt versus active polling. A simple case study shows the potential benefits of this approach. Let us consider a functional pipeline of eight matrix multiplication tasks. Each stage of this pipeline takes a matrix as input, multiplies it with a local matrix and passes the result to the next stage. We iterate the pipeline twenty times. We run the benchmark respectively on an architecture with eight and four processors. In the first case, only one task is executed on each processor, while in the second we added concurrency by mapping two tasks to each core. First, we compare three different configurations of the message-oriented architecture (Table 5.3). We execute the

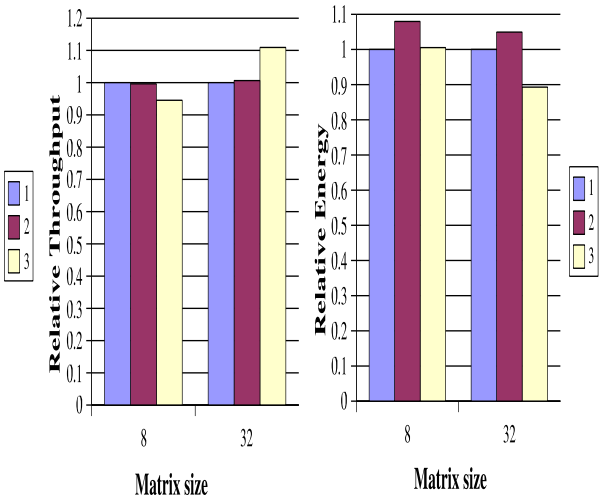


Figure 5.4: Comparison of message passing implementations in a pipelined benchmark with 8 cores from Tab. 5.3

pipeline for two matrix sizes: 8x8 and 32x32 elements. In the latter case, longer messages are transmitted.

Solution	Queue Position	Transfer Mode	Arrival Notification
(1)	scratch-queue	processor	polling
(2)	scratch-queue	processor	interrupt
(3)	scratch-queue	DMA	polling

Table 5.3: Different message passing implementations

Analyzing the results in Figure 5.4, referred to the case where one task runs on each processor, we can observe that a DMA is not always beneficial in terms of throughput. For small messages, the overhead for setting up the DMA transfer is not justified. In case of larger messages, the DMA-based solution outperforms processor-driven transfers. Conversely, employing a DMA always leads to an energy reduction, even if the duration of the benchmark is longer, due to a more power-efficient data transfer. Note that energy of all system components (DMA included) is accounted for in the energy plot. Results have been derived through functional simulation and technology homogeneous power models (0.13um technology).

Furthermore, the way a consumer is notified of the arrival of a message plays an important role, performance- and energy-wise. The consumer has to wait until the producer releases the consumer's local semaphore. With a single task per processor (Figure 5.4), the overhead related to the interrupt routine can slow down the system, depending on the communication vs computation ratio and polling is, in general, more efficient. On the contrary, with two tasks

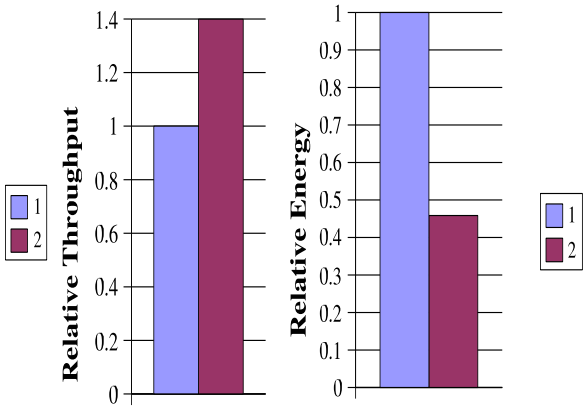


Figure 5.5: Task scheduling impact on synchronization in a pipelined benchmark with 4 cores from Tab. 5.3

per processor (Figure 5.5, referred to matrices of 8x8 elements) the interrupt-based approach performs better. In this case, it is more convenient to suspend the task because the concurrent task scheduled on the same processor is in "ready" state. Instead, with active polling, the processor is stalled and the other task cannot be scheduled.

From this example, we thus conclude that in order to optimize the energy and the throughput, the implementation of message passing should be matched with application's workload characteristics. This is only feasible by deploying a flexible message passing library.

5.6 First level classifications in the software domanin

Given the two complete and optimized hardware-software architectures for the shared memory and the message passing platforms, we now put them at work and try to capture which application characteristics and mapping decisions determine their relative performance and energy dissipation. The ultimate objective is to identify design guidelines.

Our next step in this direction is to provide a first-level classification in the software domain. We try to capture some relevant application features that can make the difference in discriminating between programming paradigms. We recall that we are targeting parallel applications, and in particular the multimedia and signal processing application domain. Relevant application features

are as follows:

- **Workload allocation policy**. It determines the way a parallel workload is assigned to the parallel computation units for the processing stage. For the class of applications we are targeting, there are two main policies:

  1. **Master-Slave paradigm**. The volume of data processed by each computation resource is reduced by splitting it among multiple slave tasks operating in a coordinated fashion. A master task is usually in charge of pre-processing data, activating slave operation and of synchronizing the whole system. Workload splitting can be irregular or regular [90]. Horizontal, vertical and cross-slicing are well-known examples of regular data partitioning, for use in video decoding. From an energy viewpoint, the benefits from shortening execution time might be counterbalanced by the higher number of operating processors, thus giving rise to a non-trivial trade-off between application speed-up and overall energy dissipation [93].
  2. **Pipelining**. Pipelining is a traditional solution for throughput constrained systems [91]. Each pipelined application consists of a sequence of computation stages, wherein a number of identical tasks are performed, executing on disjoint sets of input data. Computation at each stage may be performed by specialized application-specific components or by homogeneous cores. Many embedded signal processing applications follow this parallelization pattern [92].

- **The degree of data sharing** among concurrent tasks. Slave tasks may have to process data sets that are common to other concurrent tasks, as is the case of the reference frame for motion compensation in parallel video decoding. To the limit, all processing data could be needed by all slaves. In this case, a shared memory programming paradigm relies on the availability of shared processing data in shared memory at the cost of increased memory contention. On the contrary, employing message passing on a distributed architecture for this case would give rise to a multicast communication pattern having the master processor as source of processing data and the slave processors as the receivers. Finding the most efficient solution from a performance and energy viewpoint is again a non-trivial issue. Cache coherence support is also critical. For instance, our shared memory architecture can largely reduce the overhead for keeping shared data coherent. If a task changes shared data, it has to update/notify all other tasks with whom it shares the data. On a shared memory architecture, slaves can snoop the useful updates directly from

the shared bus, thus avoiding the transmission of updates to all tasks, which would congest the network and slow down program execution.

- The **Granularity** of processing data. Signal processing pipelines might operate on data units as small as single pixels (e.g., pixel-level video graphics pipelines) and as large as entire frames. An increased data granularity has a different impact on the volumes of traffic to be moved across the bus based on the chosen application coding style. A somewhat higher communication cost should be traded-off with the advantages given by other architectural mechanisms (e.g., data cacheability). Our exploration framework aims at spanning this trade-off and at identifying the low-level effects that come into play to determine it.

- **Data Locality.** Optimizing for data locality has been the main focus of many studies in the last three decades or so [71]. While locality optimization efforts span a very large spectrum, ranging from cache locality to memory locality to communication locality, one can identify a common goal behind them: maximizing the reuse of data in nearby locations, i.e., minimizing the number of accesses to data in far locations. There have been numerous abstractions and paradigms developed in the past to capture the data reuse information and exploit it for enhancing data locality. In this work, we refer to data locality when a piece of data is still in a cache upon reuse. Many embedded image and video processing applications operate on large multi-dimensional arrays of signals using multi-level nested loops. An important feature of these codes is the regularity in data accesses, which can be exploited using an optimizing compiler to improve cache memory performance[69]. In contrast, many scientific applications require sparse data structures and demonstrate irregular data access patterns, thus resulting in poor data locality[70].

- **Computation-to-communication ratio.** This ratio provides an indication about the communication overhead with respect to the overall computation time. In general, when this ratio is such to be haevier on the communication side, than bandwidth issues become critical to determine system performance. A good computation-to-communication ratio, together with the minimization of load imbalance, is the requirement of scalable parallel algorithms in the parallel computing domain. Hiding communication during computation is the most straighforward way to reduce the weight of communication, but other techniques can be used such as message compression or smart mapping strategies.

We now experimentally examine how the above application features influence the choice between message passing and shared memory coding styles.

Our approach is to make highly accurate comparisons of a few representative design points in the software domain, rather than making abstract comparisons covering a wide space at the cost of limited accuracy. Accuracy of our analysis will be ensured by our timing-accurate modelling and simulation environment. Varying hardware and software parameters in the considered design points will allow us to take stable conclusions and to point out power-performance trade-offs.

Our exploration space is depicted in Fig. 5.6. We split the software space based on the workload allocation policy and the degree of sharing of processing data. We aim at performing an accurate comparison of programming paradigms within the identified space partitions. Our investigations within each sub-space will take into account other application parameters such as data granularity, computation/communication ratio and data locality. To analyze each software sub-space, we have designed a set of representative and parameterizable parallel benchmarks. These latter consist of several kernels which can be typically found inside embedded system applications: matrix manipulations (such as addition and multiplication), encryption engines and signal processing pipelines. Handling parameterizable application kernels instead of entire applications provides us with the flexibility to vary computation as well as communication parameters of the parallel software, thus extending the scope of our analysis and making our conclusions more stable. Such flexibility for space exploration is frequently not allowed by complete real-life applications. Each kernel has been mapped using both the shared memory and the message passing coding style. Interestingly, the code has been deeply optimized for each programming paradigm, for a fair and realistic comparison.

- **Benchmark I- Parallel Matrix Multiplication**. A matrix multiplication algorithm was partitioned sticking to the master-slave paradigm. It was chosen to allow the analysis of applications wherein processing data is shared among the slave processors. In fact, each slave processor uses half entire source matrices and produces a slice of the result matrix (Fig. 5.7). All slices are composed together by the master processor, which is then in charge of reactivating the slave processors for a new iteration. This program is developed so as to maximize the sharing of the read-only variables (the source matrices) and to minimize the sharing of the variables that need to be updated. The size of the matrices can be arbitrarily set. A master-driven barrier synchronization mechanism is required to allow a new parallel computation to start only once the previous one (i.e., processing at all the slave processors) has completed. Overall, we simulated 5 processors: one producer and 4 slaves.

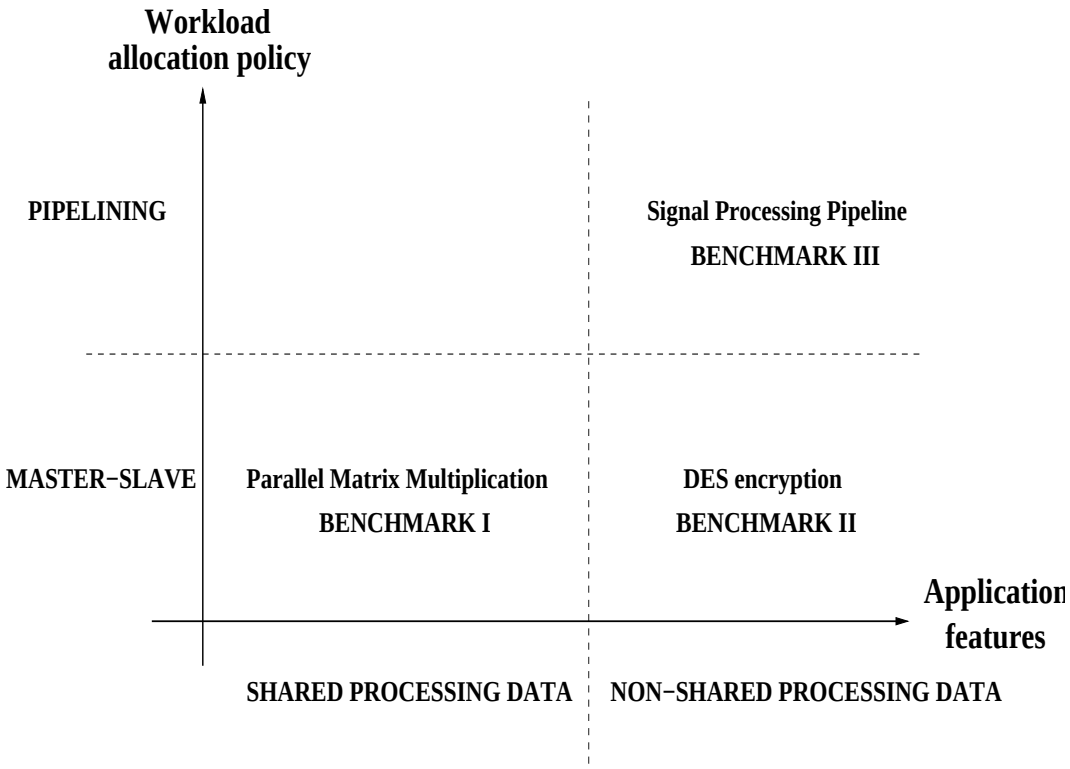


Figure 5.6: Exploration Space. Within each space partition, other software parameters have been explored such as data locality, computation/communication ratio and data granularity.

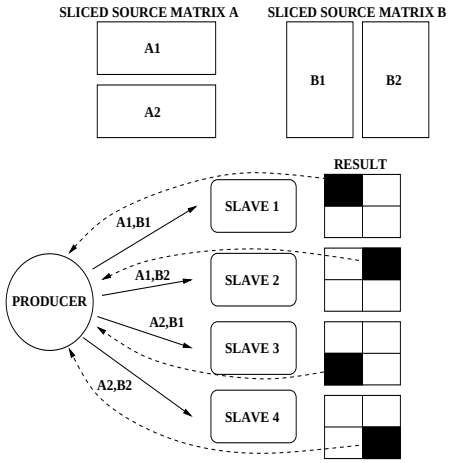


Figure 5.7: Workload allocation policies for parallel matrix multiplication.

- **Benchmark II - DES encryption**. DES (Data Encryption Standard) algorithm was chosen as an example of application that easily 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

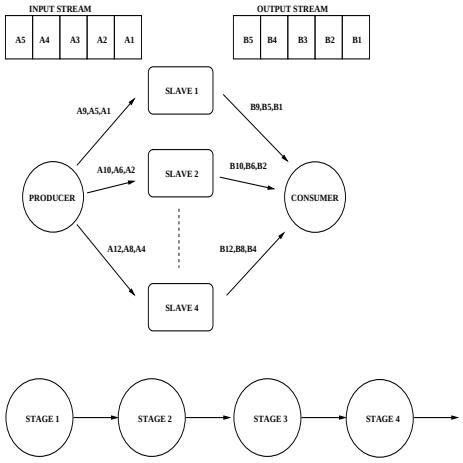


Figure 5.8: Workload allocation policy for DES encryption algorithm (up) and signal processing pipeline (bottom).

stream of 64-bit ciphered blocks. Since *each input element is independently encrypted from all others*, the algorithm can be easily parallelized. An initiator task dispatches 64-bit blocks together with a 64-bit key to n calculator tasks for encryption (Fig. 5.8-up). A collector tasks does exist, which rebuilds an output stream by concatenating the ciphered blocks of text from the calculator tasks. Please note that computation at each slave task is completely independent, since the sets of input data are completely disjoint. We modified the benchmark so to increase the size of exchange data units to multiples of 64 bits, thus exploring different data granularities. Here slave tasks just need to be independently synchronized with the producer, which alternatively provides input data to all of the slaves, and with the collector task. In this benchmark, no shared data exists. Overall, we simulated 6 processors: the producer, the consumer and 4 slaves.

- **Benchmark III - Signal Processing Pipeline**. This application consists of several signal processing tasks executing in a pipelined fashion. Each processor computes a two dimensional filtering task (which in practice reduces to matrix multiplications) and feeds its output to the next processor in the pipeline. All pipeline stages perform computations on disjoint sets of input data, as depicted in Fig. 5.8-bottom. Synchronization mechanisms (interrupts and/or semaphores) were used for correct data propagation across the pipeline stages. We simulated an 8-stages signal processing chain. For the pipeline-based workload allocation policy, we did not explore the case of processing data shared among the pipeline stages, because we consider it to be of minor interest for the multimedia

domain.

We have optimized the code of these benchmarks for both the shared memory and the message passing paradigm, as hereafter described. When using the message passing library, we always selected the active polling configuration, since we always run single tasks per processor. In this context, interrupts do not result in a better resource utilization, but only in scheduling overhead. Moreover, in our comparison with shared memory, we used the best message passing performance result, which was given sometimes by using DMA and some other times by using processor-driven transfers.

Moreover, since the system interconnect is a shared bus, we expect the update-based cache coherence protocol to have an advantage over invalidate-based one. In fact, when the producer writes data to shared memory, and those data are in the caches of other cores, this data is directly updated without further bus transactions. This inherent broadcasting mechanism brings even more advantages when many data blocks are shared among slave processors. For these reasons, we use the update protocol, in contrast to many previous papers targeting parallel computers [99].

Finally, in order to eliminate the impact of I/O from benchmark execution (this aspect is outside the scope of our analysis), we assume that input data is stored on an on-chip memory, from where it is moved or accessed according to the programming style.

5.7 Experimental results

In this section, we examine how the application characteristics and mapping decisions influence the performance and energy ratio between shared memory and message passing. First, we explain the simulation framework in which these experiments are conducted.

5.7.1 Simulation framework

Our experimental framework was based on the MPARM simulation environment [96], which performs functional, cycle-true simulation of ARM-based multi-processor systems. This level of accuracy is particularly important for MPSoC platforms, where small architectural features might determine macroscopic performance differences. Of course, simulation accuracy has to be traded off with simulation performance (up to 200000 cycles/sec with the MPARM platform). MPARM makes available a complete analysis toolkit allowing to monitor performance and energy dissipation (based on industry-provided power models) of platform components for the execution of software

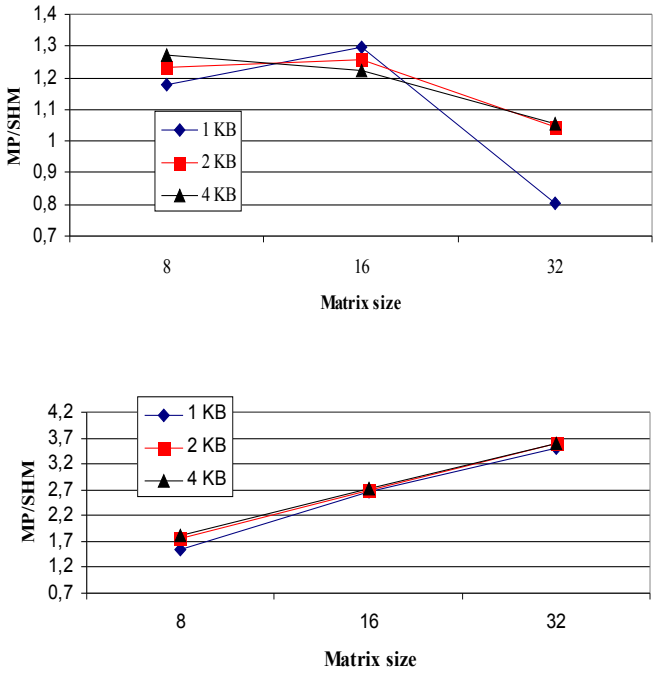


Figure 5.9: Execution time ratio. D-cache size is a parameter. (a) MM benchmark. (b) synth-MM benchmark.

routines as well as of an entire benchmark. Simulation is cycle accurate and bus-signal accurate. Our virtual platform leverages technology-homogeneous (0.13 um) power models of all system components (processor cores, system interconnect, memory devices) provided by STMicroelectronics [97, 98]. Processor core models take into account the cache power dissipation, which accounts for a large fraction of overall power.

5.7.2 Master–Slave, Shared Data

We ran the parallel matrix multiply (MM) benchmark with varying matrix size and D-cache size and for the two different hardware-software architectures. We measured the execution time for processing 20 matrices. Then, we modified the benchmark so to perform sum of matrices instead of multiplications (synthetic benchmark, $synth - MM$), thus exploring the computation versus communication ratio.

Results are reported in Figure 5.9; the y-axis represents the ratio between the execution times of the benchmark in the message passing (MP) and in the shared memory (SHM) version. Plot (a) refers to the MM benchmark, while Plot (b) to synth-MM. In the diagrams, values greater than 1 denote thus a better performance (shorter execution time) of shared memory over message

Component	*Energy % over the total system energy*
Core	36%
Instruction cache	60%
Data cache	4%

Table 5.4: Energy breakdown for the shared memory platform with Matrix size 32, Data Cache size 4KB (4-way set associative), Instruction Cache size 4KB (Direct Mapped).

passing. The scratchpad was sized big enough to contain the largest processing data, since this involved realistic cuts (8kB) while playing only a marginal role in energy dissipation. The benchmark has a good data locality, therefore we expect shared memory to be effective in this case. Furthermore, with message passing, shared data blocks have to be sent to the slave processors as explicitly replicated messages, thus originating a communication overhead. Our simulation runs confirm these intuitions only partially, as depicted in Figure 5.9-(a). We observe that as we increase data size, a corresponding increase in data cache misses affects shared memory performance, thus making message passing competitive. This loss of performance can be restored by increasing the cache size. In the plot we show that the performance ratio goes back above 1 with cache sizes of 4kB. The same ratio can be actually obtained with 8kB caches, even if a fully associative cache is instantiated. This saturation point is clearly related to the matrix size.

However, with large matrices, the advantage of shared memory over message passing decreases with respect to smaller matrices: since the computational load of the MM benchmark increases more than its communication load (the computation has $O(N^3)$ complexity while the communication load is only $O(N^2)$, where N indicates the matrix size), message passing leverages its advantage of performing the computation on a more efficient memory (the scratch-pad), thus making up for the communication overhead. In general, with larger matrices the performance of message passing and shared memory tend to converge, provided the cache and the scratchpad sizes can be arbitrarily increased to deal with larger data sets.

In the rightmost point of Figure 5.9-(a), the designer has to decide whether it is more convenient to increase the cache size and to have shared memory outperforming message passing or to adopt the message passing paradigm. Since the energy plots for the two programming paradigms exhibit the same trend of Fig. 5.9 (and therefore we have not reported them), we can take two conclusions. First, increasing the cache size to 4kB with matrix size 32 makes shared memory not only more performance-efficient, but also more energy-efficient. The reason can be deduced from Table5.4: in this case, the data cache energy is almost negligible with respect to instruction cache and processor contributions.

Therefore, a larger data cache reduces cache misses and hence application execution times in this context.

With the synth-MM benchmark (Figure 5.9-(b)), the ratio between the computational load and the communication one does not vary with the size of the data; therefore, the communication overhead of the message passing solution increases with respect to the shared memory version, where there is no need to move data. The same trend is followed by the energy curves, and is therefore not reported for lack of space.

For the shared memory version of the MM and synth-MM we reported only results of the cache-coherent platform, due to the poor performance showed by the non-coherent platform.

5.7.3 Master–Slave, Non-Shared Data

In this experiment, we ran the DES benchmark in the message passing and shared memory versions, for varying granularity of processing data. In this case, computation complexity is similar to synth-MM benchmarks, and this might lead to the conclusion that shared memory is the right choice here. However, this benchmark emphasizes also other features that put previous conclusions in discussion.

First, this is a synchronization-intensive benchmark, and previous work in the parallel computing domain agrees on the fact that performing synchronization by means of shared memory variables is inherently inefficient[85]. However, this disadvantage of shared memory over message passing (which can exploit the synchronization implicit in the arrival of a message) can be counterbalanced by using interrupt-based synchronization. The issue is to find out whether, in an MPSoC domain, using interrupts in a shared memory system is more costly than the mechanism used to wait for messages in a message passing implementation.

Second, a static profiling of the DES benchmark points out poor data locality. Similarly, many scientific applications do not exhibit much temporal locality, as all or most of the application data set is rewritten on each iteration of the algorithm. Finally, DES input data sets for each processor are disjoint, thus minimizing the advantage of using update-based cache coherence protocols. It is difficult to predict how the above features combine to determine final performance and energy metrics in the MPSoC domain, thus motivating our simulation-based analysis. Results for the DES benchmark are reported in Fig. 5.10.

At first, let us observe the relevant impact of synchronization on performance. On one hand, it causes throughput to increase as the size of exchanged

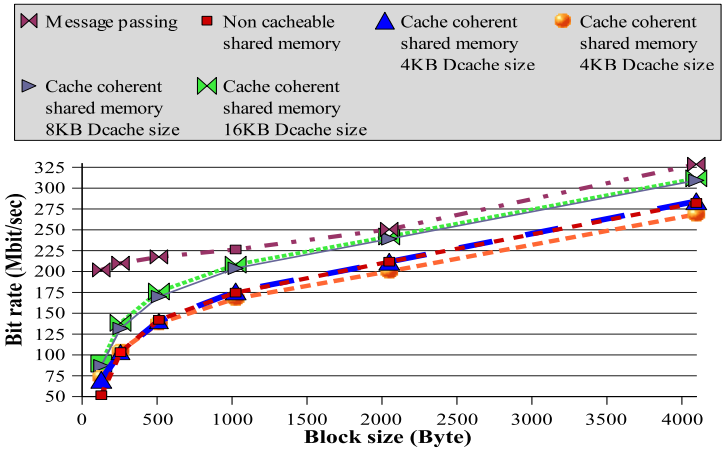


Figure 5.10: Throughput for the DES benchmark as a function of data granularity.

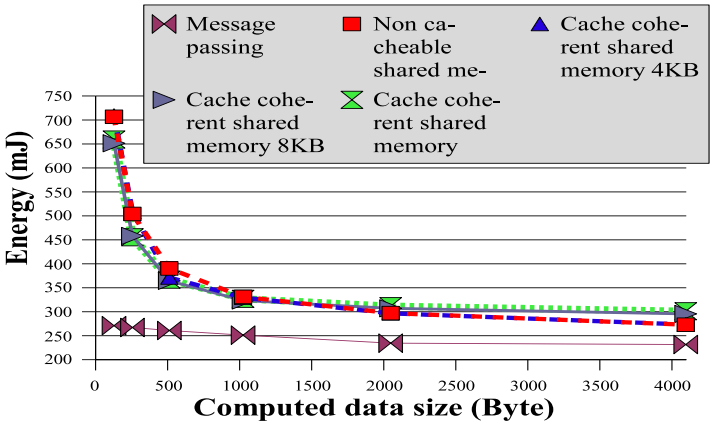


Figure 5.11: Energy for the DES benchmark as a function of data granularity.

data units increases. In fact, processors still elaborate the same overall amount of data, but they exchange data units with larger granularity, thus incurring fewer synchronization events. Please note that the increase in communication translates into a linear increase of computation, thus resulting in the linear increase of throughput.

On the other hand, for small data units, shared memory scales worse than message passing due to the high overhead associated with interrupt handling. In fact, the idle task is scheduled to avoid polling remote semaphores, and the DES task is re-scheduled when an interrupt is received. On the contrary, message pasing can poll a distributed local semaphore without accessing the bus. This inefficiency incurred by shared memory significantly impacts its performance with respect to that of message passing, which is clearly the best solution for small data units.

In addition, Fig. 5.10 also shows that the message passing approach clearly outperforms shared memory over all the range of explored data granularity. Unlike the synth-MM benchmarks, where a larger data size results in an increasing efficiency of shared memory over message passing, here the advantage of message passing over shared memory does not reduce but stays constant over the range of explored data unit size.

In fact, as data footprint increases, the lower synchronization overhead of shared memory is progressively counterbalanced by the increasing cache miss ratio of the consumer processor, and the two low level effects compensate each other, as showed by the parallel curves in Fig. 5.10.

In this case, the degrading data cache performance is not related to cache conflicts, but rather to the limited cache size. In fact, as Fig. 5.10 indicates, a fully associative cache provides negligigle performance benefits. On the contrary, shared memory performance can be significantly improved by increasing the data cache size from (default) 4kB to 8kB. The underlying reason is that while the cache miss ratio of all slave processors stays constant as data size increases, this does not hold for the consumer. This latter reads slave output data from shared memory. While for small data units the corresponding memory locations can be contained in the consumer cache without conflicts, a larger data footprint causes an increasing number of conflicts in the 4kB data cache (from 4 to 11%), that penalizes shared memory.

Interestingly, further increasing the data cache size from 8kB to 16kB leads to a performance saturation effect, which indicates that in this scenario a message passing solution is inherently more effective. Moreover, reverting to such large caches starts impacting also system energy, as illustrated in Fig. 5.11. The trend of energy curves is strongly correlated to the performance plot, in that a higher throughput determines a shorter execution time to process the same amount of data.

5.7.4 Pipelining

We finally ran the pipelined matrix processing benchmarks (multiplication and addition), and reported simulation results in Fig. 5.12.

Consider case (a), i.e. matrix multiply. This benchmark has features common to both MM and DES benchmarks. Like MM, here we have high data locality and high computation complexity. Like DES, we have a high impact of synchronization mechanisms. Results show that for small matrices, the more efficient synchronization carried out by message passing is compensated by the higher time spent for inter-processor communication: with shared memory, cache updates occur in parallel with task execution, while with message

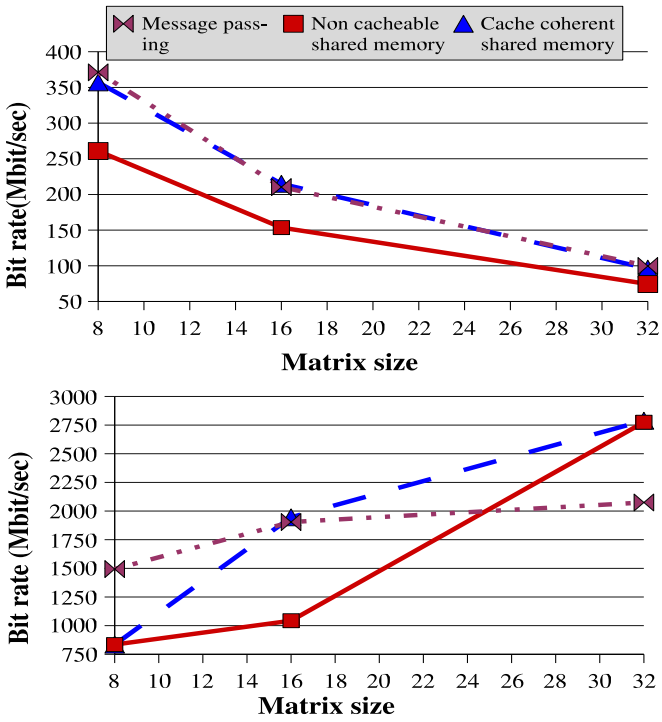


Figure 5.12: Throughput for pipelined matrix processing. (a) Matrix multiplication. (b) Matrix addition.

passing the small data size is not in favor of using a DMA due to the programming overhead. Pros and cons of each paradigm compensate each other and we do not observe any performance difference.

Although counterintuitive, if matrices become large, the higher computation efficiency of message passing (shared memory incurs a significant cache miss ratio) does not determine an overall better performance of message passing. In fact, since the pipeline stages are almost perfectly balanced, all data transfers between pairs of communicating processors occur in parallel at the same time, thus creating localized peaks of bus congestion that increase transfer times. This explains the similar performance of message passing and shared memory also for large data.

In (b), the shared memory solution outperforms the message passing one as matrix size increases, reflecting what we have already seen in the synth-MM benchmark. However, if matrices are small, the high synchronization efficiency of message passing generates performance benefits, as seen for DES. Moreover, in the rightmost part of the plot we can see that cache-coherent shared memory and non cache-coherent shared memory tend to have the same performance. In fact, cache-coherent shared memory suffers from a high percentage of cache

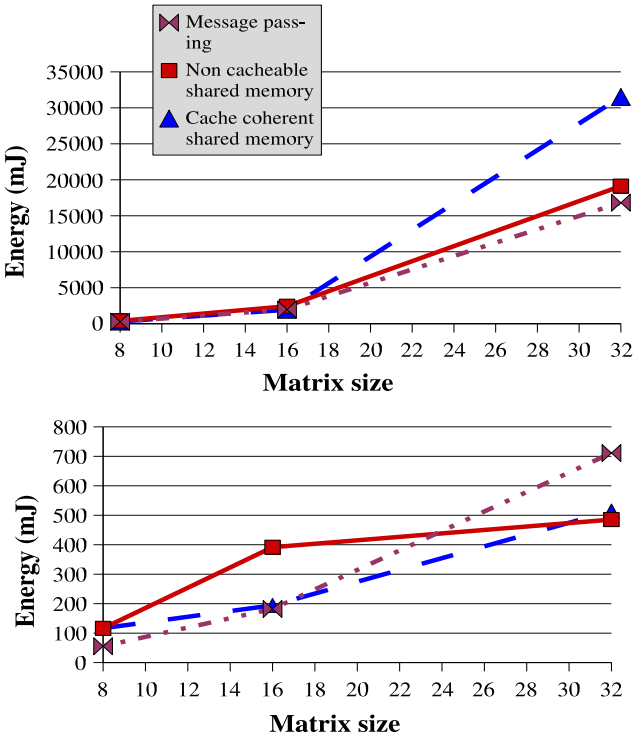


Figure 5.13: Energy for pipelined matrix processing. (a) Matrix multiplication. (b) Matrix addition.

Task	1	2	3	4	5	6	7	8
Cycles	19497	56109	53497	112095	28281	46848	18973	19432

Table 5.5: The computation cost of each task of the pipeline

Processor	1	2	3	4
Task Mapping 1	6,7	4,5	3	0,1,2
Task Mapping 2	2,3	6,7	4,5	0,1

Table 5.6: Mapping of tasks on the processors

misses, and this counterbalances the more efficient accesses to shared memory.

In Fig. 5.13 (a) we see that the shared memory variant consumes more energy, since we have an increase of data cache misses. On the contrary, in (b) communication plays a more significant role, therefore message passing progressively becomes less energy-efficient.

Impact of mapping decisions

For balanced pipelines, message passing suffers from the high peak bandwidth utilization problem that limits its performance. Let us now show that this lim-

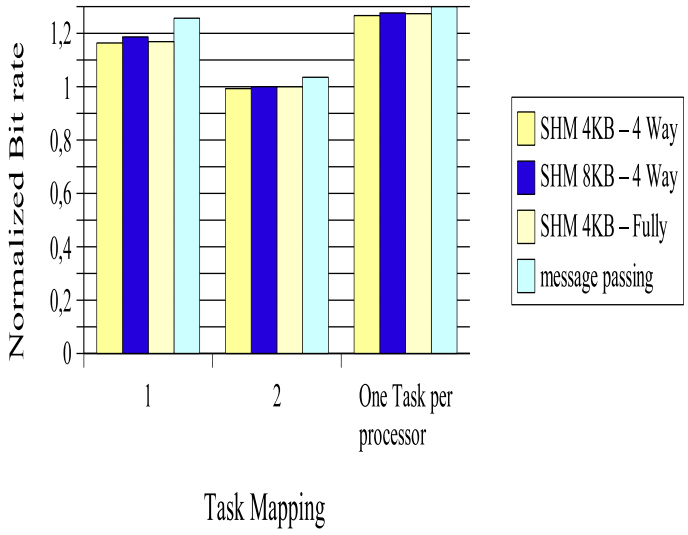


Figure 5.14: Bit rate achieved with the different mappings.

itation can be relieved by taking the proper course of actions, and that the performance that can be achieved in this way cannot be achieved by shared memory by varying cache settings. We consider a pipeline of matrix multiplications, where a different number of operations is performed at each stage, thus making the pipeline unbalanced (see Table5.5). The rightmost bars in Fig.5.14 indicate that message passing outperforms shared memory in this context, even though the difference is not significant. However, if a lower throughput is needed, by rearranging task allocation to processors and allowing more tasks to run on the same processor, we can get a more noticeable differentiation between message passing and shared memory, provided communication is taken into account in the mapping framework. We focused on a 500 MBit/sec target throughput, and considered two mappings that meet the performance constraint while generating different amounts of bus traffic. The mappings are reported in Table5.6, and the first one was communication-optimized by using the framework in[110]. By looking at the results in Fig.5.14, the message passing implementation of mapping 1 outperforms that of mapping 2. The performance difference can be explained by the peaks in bandwidth utilization, which increase the time spent in transferring data. Finally, the plot shows that shared memory performance is always lower than that of message passing, whatever the cache configuration (size and associativity), thus proving a higher efficiency of message passing for this context.

5.8 Contrasting programming paradigms for MPSoCs and parallel computers

Our exploration has pointed out some main differences between programming paradigms for MPSoCs with respect to those for the parallel computing domain. We summarize them as follows:

- In shared memory platforms, the use of shared buses makes update-based cache coherence protocols effective for producer-consumer communication, without generating traffic overhead as is the case for many network-centric parallel computer architectures. Furthermore, caches tend to smooth the distribution of data traffic, hence reducing the probability of traffic peaks on the interconnect.

- MPSoCs have access to a fast communication architecture integrated on the die together with the processors. As a result, memory can be accessed faster and thus the cache-lines can be refilled more easily than on a traditional multiprocessor architecture. In practice, this also means that on an MPSoC the same performance can be obtained with a smaller cache, even if this causes cache misses to increase. The latter insight is often used by designers to reduce chip area and thus manufacturing cost. However, if the bandwidth of the communication architecture becomes congested, the communication delay increases again and the extra cache-misses then result in a high performance loss and in a system energy overhead associated with longer execution times. Hence, even though with a smaller cache we can obtain the same performance, the smaller cache makes the performance more sensitive to bus congestion, potentially limiting the efficiency of shared memory.

- In the MPSoC context, the software infrastructure is far more lightweight than in traditional parallel systems. Therefore, many performance overhead sources that have been traditionally considered negligible or marginal, now come into play and in some cases might make the difference. Two relevant examples that have emerged throughout this work are the overhead for DMA programming (which must be compared with the size of data to move) and for interrupt handling (to be compared with the bus congestion induced by semaphore polling). Surprisingly, solutions that are apparently inefficient might turn out to provide the best performance, such as processor-driven data transfers and polling-based synchronization. A similar issue concerns porting of standard messaging libraries on MPSoC platforms. The porting process of these libraries (such

as the SystemV IPC library considered in this work or the MPI primitives) has to be combined with an optimization and customization effort to the platform instance in order to reduce its performance overhead. As an example, the several thousands cycles latency incurred by MPI primitives [102] in traditional parallel systems would seriously impair MPSoC performance. This further stresses the importance of hardware extensions for the different programming paradigms, as we have done in this work.

- In message passing architectures, local memories in processor nodes cannot be as large as in traditional distributed memory multiprocessor systems. On the other hand, software-controlled scratch-pad memories exhibit a negligible access cost, performance- and energy-wise. We think that this feature, combined with technology constraints in memory fabrications, will further differentiate MPSoC platforms from distributed parallel computers. We expect this to impact the architecture of the memory hierarchy, which will have to store large data sets off-chip while at the same time avoiding the bottleneck of centralized off-chip memory controllers. Considering these issues is outside the scope of this work, which has therefore assumed that processing data can be entirely contained in scratchpad memories, while keeping reasonable memory sizes.

5.9 Design guidelines

A designer can choose the architectural template and the programming paradigm that best suits its needs based on a few relevant features of the parallel application under development. Our analysis has showed the importance of *workload allocation policy*, *computation/communication ratio*, *degree of sharing of input data among working processors* and *data locality* in differentiating between the performance and energy of the message passing versus the shared memory programming paradigm. Since our approach is centered around the accuracy of the exploration framework, we restricted our analysis to three relevant scenarios for future MPSoC platforms, which were extensively and accurately investigated by means of synthetic and parameterizable benchmarks. This leads us to the following guidelines for system designers:

- For the case where many working processors share the same input processing data, shared memory typically outperforms message passing. Shared memory leverages the implicit broadcasting support offered by the write-through update cache coherence protocol. In contrast, message passing suffers from the overhead for explicitely repli-

cated input messages and for post-processing updates of shared data stored in local memories. Obviously, an application with low computation/communication ratio emphasizes shared memory efficiency. The only, non-trivial case where message passing turns out to be competitive is that of computation-intensive applications with large data sets. In fact, message passing takes profit by a more efficient computation in scratchpad memory, while the shared memory implementation starts suffering from cache misses. We have showed that shared memory performance can be restored by means of proper data cache sizing, since this has only a marginal impact on system energy. However, performance of both programming paradigms tends to converge in these operating conditions.

- For synchronization-intensive applications, message passing provides potentials for the implementation of more efficient synchronization mechanisms and hence for shorter application execution times. In particular, this point makes the difference in presence of processing data with small footprint. Synchronization events can be very costly for MPSoC systems, in terms of bus congestion for remote semaphore polling or performance overhead for interrupt handling and task switching. The frequency and duration of these events, and hence their impact on application execution metrics, depends on the amount of computation performed on each input data, on input data granularity and on relative waiting times between synchronized tasks. We have observed that this issue certainly determines better system performance and energy of message passing when small input data is to be processed in synchronization-intensive applications.

- Many applications (e.g., scientific computation, criptography) make use of iterative algorithms showing poor temporal locality, where all or most sets of input data are rewritten at each iteration of the algorithm. In this scenario, message passing turns out to be a more effective solution than shared memory, even though different cache settings might reduce the gap. The message passing solution is also the most energy-efficient.

- As regards signal processing pipelines, what really makes the difference between the two programming paradigms is the computation/communication ratio and data granularity. For small data sets, message passing again takes profit by the most efficient synchronization mechanism, which is key for pipeline implementations. On the other hand, as the data footprint increases, message passing proves slightly more effective only for computation-intensive pipeline stages. However, in this regime message passing performance is extremely sensitive to

peak bus bandwidth utilization, and for balanced pipelines or significant peak bandwidth requirements (associated with input data reading or output data generation) shared memory becomes competitive. Instead, shared memory noticeably outperforms message passing with a low computation/communication ratio and large data sets, since the communication overhead of message passing cannot be amortized by enough computation in scratchpad memory.

Chapter 6

Hardware/Software Architecture for Real-Time ECG Monitoring

6.1 Abstract

The interest in high performance chip architectures for biomedical applications is gaining a lot of research and market interest. Heart diseases remain by far the main cause of death and a challenging problem for biomedical engineers to monitor and analyze. Electrocardiography (ECG) is an essential practice in heart medicine. However, ECG analysis still faces computational challenges, especially when 12 lead signals are to be analyzed in parallel, in real time, and under increasing sampling frequencies. Another challenge is the analysis of huge amounts of data that may grow to days of recordings. Nowadays, doctors use eyeball monitoring of the 12-lead ECG paper readout, which may seriously impair analysis accuracy. Our solution leverages the advance in multi-processor system-on-chip architectures, and it is centered on the parallelization of the ECG computation kernel. Our Hardware-Software (HW/SW) Multi-Processor System-on-Chip (MPSoC) design improves upon state-of-the-art mostly for its capability to perform real-time analysis of input data, leveraging the computation horsepower provided by many concurrent DSPs, more accurate diagnosis of cardiac diseases, and prompter reaction to abnormal heart alterations. The design methodology to go from the 12-lead ECG application specification to the final HW/SW architecture is the focus of this paper. We explore the design space by considering a number of hardware and software architectural variants, and deploy industrial components to build up the sys-

tem.

6.2 Introduction

Despite the ongoing advances in heart treatment, in the United States [113] and Canada [114] as well as in many other countries, the various forms of cardiovascular disease (CVD) and stroke remain by far the number one cause of death for both men and women regardless of ethnic backgrounds. According to the World Health Organization (WHO) Report in 2003, 29.2% of total global deaths are due to CVD, many of which are preventable by action on the major primary risk factors and with proper monitoring [113]. It is estimated that by 2010, CVD will be the leading cause of death in developing countries. Since the rate of hospitalization increases with age for all cardiac diseases [115], a periodic cardiac examination is recommended. Hence, more efficient methods of cardiac diagnosis are desired to meet the great demand on heart examinations. However, state-of-the-art biomedical equipment for heartbeat sensing and monitoring lacks the ability of providing large-scale analysis and remote, real-time computation at the patient's location (point of need). The intention of this work is to use MPSoC microelectronic technology to meet the growing demand for telemedicine services, especially in the mobile environment. The project attempts to address the existing problem of reducing the costs for hospitals/medical-centers through using MPSoC-based designs that may replace biomedical machines and have higher quality, reduce the nurse's and doctor's work-load, and improve the quality of healthcare for patients suffering from heart diseases by exploring one potential solution. From the hospital side, deploying this solution will further reduce the costs of rehabilitating and following up on patients "primary care" since it allows better home-care. Home-care ensures continuity of care, reduces hospitalization costs, and enables patients to have a quicker return to their normal life styles. From a technical viewpoint, real-time processing of ECG data would allow a finer-granularity analysis with respect to the traditional eyeball monitoring of the paper ECG readout. Eventually, warning or alarm signals could be generated by the monitoring device and transmitted to the healthcare center via telemedicine links, thus allowing for a prompter reaction of the medical staff. In contrast, heartbeat monitoring and data processing are traditionally performed at the hospital, and for long monitoring periods a huge amount of collected data must be processed offline by networks of parallel computers. New models of healthcare delivery [114] are therefore required, improving productivity and access to care, controlling costs, and improving clinical outcomes. This poses new technical challenges to the design of biomedical ECG

equipment, calling for the development of new integrated circuits featuring increased energy efficiency while providing higher computation capabilities. The fast evolution of biomedical sensors and the trend in embedded computing are progressively making this new scenario technically feasible. Sensors today exhibit smaller size, increased energy efficiency and therefore prolonged lifetimes (up to 24 hours) [116], higher sampling frequencies (up to 10 kHz for ECG) and often provide for wireless connectivity. Unfortunately, a mismatch exists between advances in sensor technology and the capabilities of state-of-the-art heart analyzers [117], [118], [119]. They cannot usually keep up with the data acquisition rate, and are usually wall-plugged, thus preventing for mobile monitoring. On the contrary, the deployment of wearable devices such as SoC devices has to cope with the tight power budgets of such devices, potentially cutting down on the maximum achievable monitoring period. In this paper we propose a wearable multi-processor biomedical-chip for electrocardiogram (MPSoC ECG biochip) paving the way for portable real-time electrocardiography applications targeting heart disorders. The biochip leverages the computation horsepower provided by many (up to twelve) concurrent DSPs and is able to operate in real-time while performing the finest granularity analysis as specified by the ECG application. Moreover, in case of heart failure emergency aid should arrive in a period of few minutes from the time when the heart failed, otherwise brain damage may occur. Hence, real time analysis must be done in few seconds to allow the alarm signal to reach the emergency aid team, which should act immediately. The biochip system builds upon some of the most advanced industrial components for MPSoC design (multi-issue VLIW DSPs, high-throughput system interconnect and commercial off-the-shelf biomedical sensors), which have been composed in a scalable and flexible platform. Therefore, we have ensured its reusability for future generations of ECG analysis algorithms and its suitability for porting of other biomedical applications, in particular those collecting input data from wired/wireless sensor networks [120]. The paper goes through all the steps of the design process, from application functional specification to hardware modeling and optimization. System performance has been validated through functional, timing accurate simulation on a virtual platform. We point out the need for simulation abstractions matching the application domain. A 0.13μm technology-homogeneous power estimation framework leveraging industrial power models is used for power management considerations [97], [98]. The paper presents the process of software functional specification, optimization and parallelization, as well as the results of the hardware design space exploration, which leads to the final performance- and energy-optimized solution.

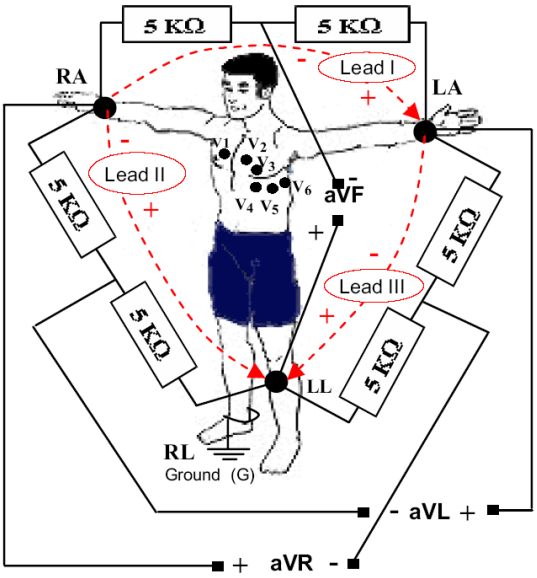


Figure 6.1: 12-lead ECG: RA, LA, LL, & RL are the right arm, left arm, left leg, and right leg sensors. RL is grounded (G).

6.3 Biomedical Background

The electrocardiogram (ECG) is an electrical recording of the heart activity that is used as a diagnosis tool by physicians and doctors to check the status of the heart. The most commonly used way to detect the heart status is the 12-lead ECG technique. This technique uses nine sensors on the patient's body 6.1. The three main sensors are distributed by: placing one sensor on the left arm (LA), a second sensor on the right arm (RA), and a third sensor on the left leg (LL). The right leg (RL) is connected by only a wire to be used as ground for the interconnected sensors. By only having these three sensors physicians can use a method known as the 3-lead ECG, which suffers from the lack of information about some parts of the heart but is useful for some emergency cases to have quick analysis. In this respect, medical doctors require more sensors (i.e., more leads). Hence, six more sensors (V1-V6) are added on the chest (Fig. 1). The voltages V1-V6 are measured with respect to Ground (G) on the right leg (RL). In some cases, physicians use these six chest-placed sensors to analyze the heart. Using all the nine sensors and interconnecting them for the 12-lead ECG gives twelve signals known in biomedical terms as: Lead I, Lead II, Lead III, aVR, aVL, aVF, V1, V2, V3, V4, V5, and V6 (6.1). The 12-lead ECG produces huge amounts of data especially when used for a long number of hours. Physicians use the 12-lead ECG method, because it allows them to view the heart in its three dimensional form; thus, enabling detection of any abnormality that

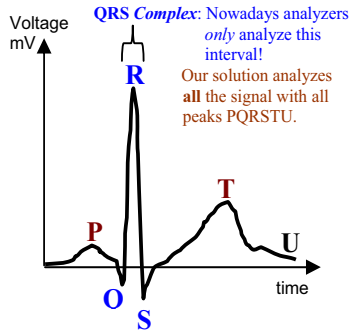


Figure 6.2: Ideal ECG Signal for lead I.

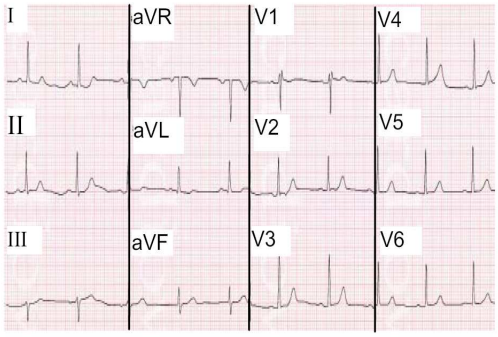


Figure 6.3: Complete paper readout, which is not accurate to see peaks nor easy to read for long recordings.

may not be apparent in the 3-lead or 6-lead ECG technique. 6.2 shows an explanatory example of a typical ECG signal. The most important points on the ECG signal are the peaks: P, Q, R, S, T, and U. Each of these peaks is related to a heart action that is of importance to the medical analysis. Figure 3 shows real recorded signals from 12-leads, which are printed on the eyeballing paper. This paper printout is the classical medical technique used for looking at ECG signals, and it is still used.

However, the eyeballing paper print makes the check of the different heart peaks and rhythms difficult and inaccurate due to its dependence on the physician's eyes. On the other hand, when using digital recording and filtering we can determine the peaks more accurately. Consequently, we can use digital computing to process the sensed data and analyze the heart beat. In addition, there are normal medical ranges for the inter-peak time intervals, and every combination of different inter-peak intervals proves a type of heart illness. The most important of the peaks is the R peak, which refers to the largest heart blood pump.

6.4 Previous Work

Electrocardiogram methods for heart analyses have been one of the most important medical practices, hence, the monitoring and analyses of ECG signals have not only gone through a lot of research work, but also many companies have investigated and worked on commercial solutions. However, we are not aware of any solution in the research or the commercial markets that is composed of a single-chip real-time analysis solution for full 12-lead ECG, and that is able to estimate the heart period independent of the peak signals and, at

the same time diagnose all the peaks: P, Q, R, S, T and U and their inter-peak intervals to result in disease diagnosis. Most of the work done involves only recording huge amounts of data in large storage media and then analyzing the stored data, but not allowing the ease of patient mobility. Most of the time, the patient has to be confined to a bed for a number of hours (could be for a whole day). Some commercial solutions are only capable of concluding if the heart beat is normal or abnormal but can not specify the period nor could they diagnose the disease. Other real time solutions available in the market, in healthcare institutes, and in research organizations, are only capable of sensing and transmitting ECG data [121] to: either a local machine [122] or to a distant healthcare center [123]. In both cases, the work that is executed involves checking if the heart beat is healthy or unhealthy without analyzing the disease and not in real-time. Moreover, the commercial solutions under study [124] do not look into the parallelization of the ECG analysis into multiple cores, so to speed up processing.

6.5 Sensing and Filtering Stage

ECG analysis requires three main phases: (i) acquiring the signals from the leads, (ii) filtering the lead-signals (each alone), and (iii) analysis 6.4. Firstly, the sensing phase requires an A/D converter in order to be able to have digital data for our digital filter. We use 16 bit A/D converters, because our analysis algorithm and ECG biochip are designed based on having 16-bit filtered data as input. We briefly discuss the filtering method we use as an essential part of our proposed solution, and then we discuss the biochip design that depends on this filtering step. The high investment in sensor technology and biomedical research in general gave the birth to biomedical sensors that have more advanced features than the commercial available ones just a few years ago. For instance, the nowadays sensors are characterized by prolonged lifetimes (up to 24 hours), and higher sampling frequencies (up to 10 kHz for ECG). Some sensor companies have produced wireless biomedical sensors in order to aid patient mobility [116]. This advance in biomedical sensors faces a mismatch with biomedical heartbeat analyzers that still lack behind to cope with the huge amounts of data, the high rates, and the wireless features that modern sensors can provide [118]. In our work, many sensors may be chosen, and for the moment we choose the sensors that can serve our real-time aim and that have reasonable prices for the market success of the solution, hence we choose the state of the art commercial sensor from Ambu Inc. silver/silver chloride B̈lue Sensor R¨ [116] shown in 6.4. It is characterized by: 24 hour lifetime, superior adhesion, optimal signal measuring during stress tests. It is small to

carry (57mm x 48mm), and it is easily wearable. On the other hand, even the state of the art sensors suffer from the usual problems that most biomedical sensors suffer from. For instance, data provided by biomedical sensors suffers from several types of noise: physiological variability of QRS complexes (The QRS Complex is shown in 6.2, baseline wander, muscle noise, artifacts due to electrode motion, power-line interference [125]. The presence of several noise sources might impair ECG analysis accuracy, as showed in the R-Peak detection marked by circled areas in 6.5. Two peaks may be detected where there should be only one. In order to deal with noisy input signals, we designed an IIR filter with order 3 that outputs its results in 16-bit binary format 6.4. However, we need to be aware of the fact that we want to look in our solution at high sampling frequencies (250Hz, 1000Hz and above), because we want to: (a) make use of the available accuracy of the state of the art sensors, (b) have finer granularity of data, and (c) get more accurate analysis since in some cases more data samples are needed to discover a disease; like, for instance, the medical case known as the R on T phenomena [126], where the R and the T peaks are very near in time so we need a very high number of samples and an intelligent algorithm to discover them. Moreover, it is extremely important to choose a sampling frequency that minimizes the risk of aliasing. The highest frequency needed for the ECG signal is 90Hz (due to the medical frequencies of the heart), which implies that the lowest sampling frequency that can be used is equal to the Nyquist rate (180Hz). However, in order to sample at such a frequency, the analogue signal has to be band limited to 90Hz, which can be achieved by the use of a complex analogue bandpass filter with a very sharp frequency response. This solution, although advantageous on limiting the amount of data to be stored, has a disadvantage on the analogue side, since the bandpass filter, being complex in order to meet the sharpness requirement, will probably have a considerable power consumption. An alternative solution would be to sample at a frequency much higher than the Nyquist rate, such that the analogue bandpass filter can have a relaxed frequency response, while still effectively filtering out the frequencies that would cause aliasing during sampling. For instance, by choosing a sampling frequency of 5kHz, all frequencies beyond 2.5kHz would have to be filtered out before sampling, but that task is simpler than before, since all frequencies between 90Hz and 2.5kHz can be attenuated without affecting the data needed for analysis. After sampling, band limitation to 90Hz can be implemented using a digital filter. This approach has the advantage of using a lower-complexity bandpass filter, and reducing considerably the risk of aliasing and folding. Moreover, increasing the number of samples increases the accuracy of the sample, and makes the overall filtered signal smoother when used for analysis. Our IIR filter is built to deal with

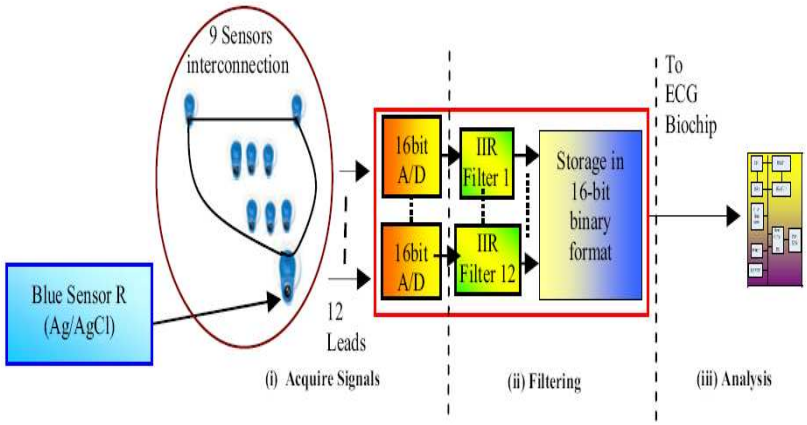


Figure 6.4: The System for sensing and filtering of ECG lead signals before sending data to the ECG Biochip for analysis. Blue Sensor R is from Ambu Inc. [116].

these problems. Another main advantage of using the IIR filter is to eliminate the noise that is directly proportional to the DC offset of the sensed ECG [125], which is around 0.1mv. The two plots in 6.5 clearly show how the filtering algorithm remedies this problem. In our implementation, the filter is implemented in hardware on a dedicated chip feeding the external SDRAM memory of our biochip. Our filter is the convolution of the noisy signal with the filter impulse response given in (1):

$$y[n] = \sum_{k=1 \rightarrow \infty} h[k] * x[n-k] \textit{ [1]}$$

where, x[n] is the noisy signal, h[n] is the filter impulse response, and n is the sample index. This filter in (1) is also an infinite impulse response (IIR, Chebyschev filter), so it can be written as (2):

$$y[n] = \sum_{l=0} x[n-l] * b[l] - \sum_{m=1} x[n-m] * a[m] \textit{ [2]}$$

where, y is the output of the filter and x is the input, b is the vector that contains the filter coefficients for signal x, and a is the vector that contains the filter coefficients for output y.

The upper limits of the coefficients are dependent on the order of the filter being used. Our IIR filter is of order 3, because our ECG data does not require higher orders. We can improve our filter (when needed) by simply knowing

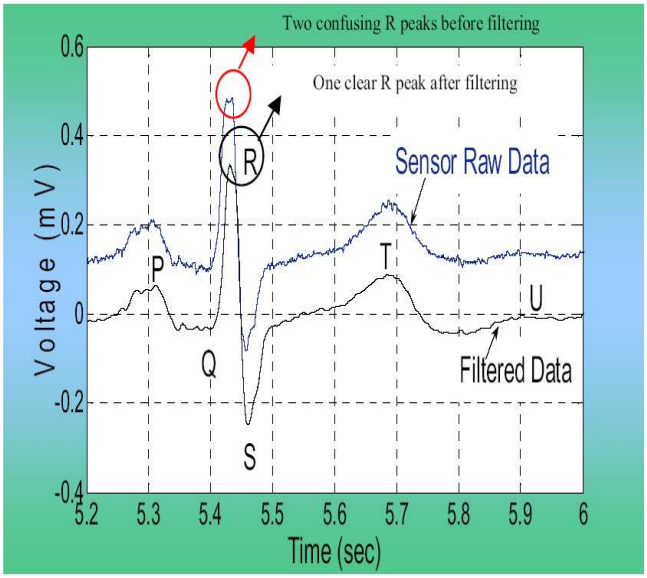


Figure 6.5: ECG raw and filtered data (lead I).

the needed values of the coefficients in vectors a[.] and b[.].

6.6 ECG Algorithm

Most ECG systems make use of the Pan-Tompkins analysis algorithm [127], which targets QRS complexes (6.2) detection and consists of the cascade of four filters: (i) band pass, (ii) differentiator, (iii) squaring operation, and (iv) a moving window integrator. In principle, traditional ECG analysis starts from a reference point in the heart cycle (the R-peak is commonly used as the reference point). As a consequence, accurate detection of the R-peak of the QRS complex is a prerequisite for the reliable functionality of ECG analyzers [127]. However, as an effect of ECG signal high variability, R-peak detection might be inaccurate. For instance, in the R on T phenomena, a T peak may be wrongly taken for an R peak, and then the R-T interval will be considered as an R-R interval, and the period will be wrong. Hence, other QRS parameters will be consequently inaccurate. As a result, traditional techniques may fail in detecting some serious heart disorders such as the R-on-T phenomenon (associated with premature ventricular complexes) [126]. Our approach takes a different perspective: instead of looking for the R-peaks and then detecting the period, we detect the period first (via autocorrelation) and then look for the peaks. We use an autocorrelation function (ACF) to calculate the heartbeat period without looking for peaks. Then, we can restrict our analysis to a time window equal

to the period and detect all peaks. Although potentially more accurate, our algorithm incurs a higher computational complexity: 3.5 million multiplications, which have been reduced to 1.75 million through a number of code (SW) optimizations. The single-chip multiprocessor architecture that will be selected for the practical implementation of the algorithm will provide the scalable computation horsepower needed for the highly accurate ECG analysis that we are targeting. The autocorrelation we use, as shown in (3), has a certain number of Lags (L) to minimize the computation for our specific application as discussed below. We validated our algorithm over several medical traces [128], [129].

$$R_y[k] = \sum_{n=\infty \rightarrow -\infty} y[n] * y[n-k] \textit{ [3]}$$

where Ry is the autocorrelation function, y is the filtered signal under study, n is the index of the signal y, and k is the number of lags of the autocorrelation (L has an effect on the performance due to the high number of multiplications). We run the experiments for n = 1250, 5000 and 50,000 relative to the sampling frequencies of 250, 1000, and 10,000Hz, respectively. In order to minimize errors and execution time we use the derivative of the ECG filtered signal since if a function is periodic then its derivative is periodic. Hence the autocorrelation function of the derivative can give the period as shown in 6.6. In order to be able to analyze ECG data in real-time and to be reactive in transmitting alarm signals to healthcare centers (in less than 1 minute), a minimum amount of acquired data has to be processed at a time without losing the validity of the results. For the heart beat period, we need at least 4 seconds of ECG data in order for the ACF to give correct results. The autocorrelation function is deployed within the algorithm shown in 6.7, which computes the required medical parameters: heart period, peaks P, Q, R, S, T, and U, and inter-peak time spans. Peak heights and inter-peak time ranging outside normal values, which indicates different kinds of diseases, are detected with our algorithm. From a functional viewpoint, the algorithm consists of two separate execution flows: one that finds the period using the autocorrelation function (process 1 in 6.7), and another one that finds the number, amplitude and time interval of the peaks in the given 4-second ECG data (process 2 in 6.7). In process 1, we firstly find the discrete derivative of the ECG signal. This will not affect the analysis since the derivative of a periodic signal is periodic with the same period. The advantage of taking the derivative, and thus adding some overhead to the code, is that the fluctuations taking place in the signal and especially those around the peaks would be reduced to a near-zero-value. Moreover, performance overhead associated with derivative calculation of the ECG signal is negligible compared to the rest of the algorithm, especially the autocorrelation

part. Finally, if the original signal is periodic, then the autocorrelation of the derivative of the signal is periodic by definition, with the same period as that of the original signal under test. In process 2, a threshold is used to find the peaks. This threshold was experimentally set to 60% of the highest peak in the given search interval.

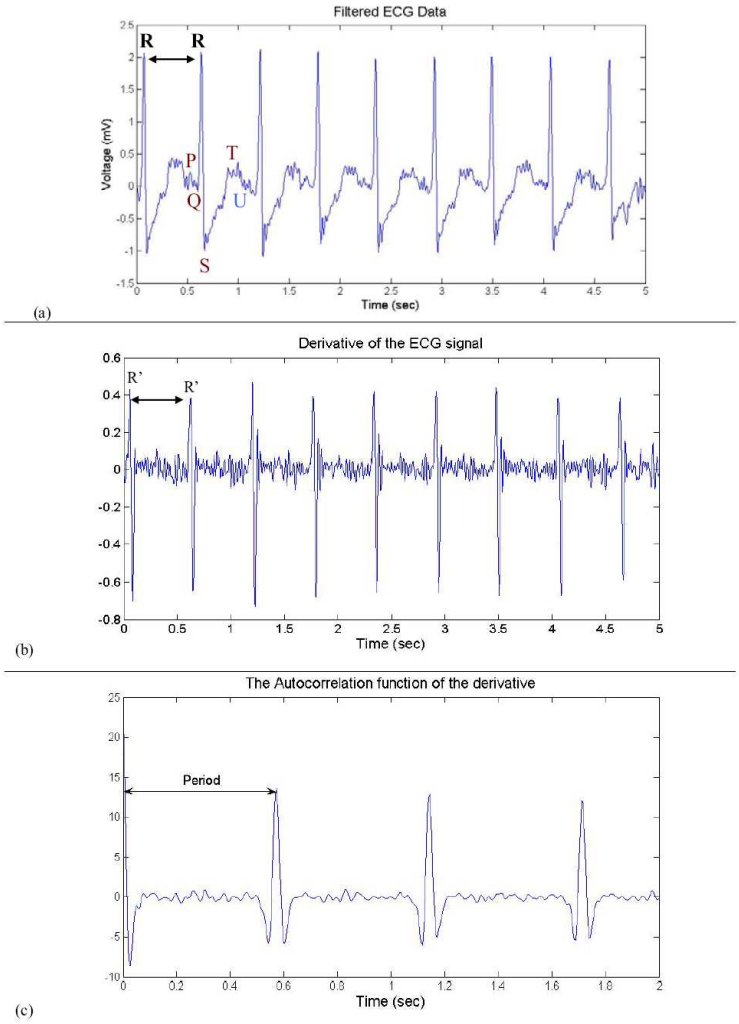


Figure 6.6: Heart period analysis: (a) ECG signal peaks P, Q, R, S, T, and U; (b) derivative amplifying R peaks; (c) autocorrelation of the derivative characterized by significant periodic peaks having the same value as the period of the ECG signal in (b) and thus (a).

Our proposed ECG-analysis algorithm was conceived to be parallel and hence scalable from the ground up. Since each lead senses and analyzes data independently, each lead can then be assigned to a different processor. So, to extend ECG analysis to 15-lead ECG or more, then what is required is to change

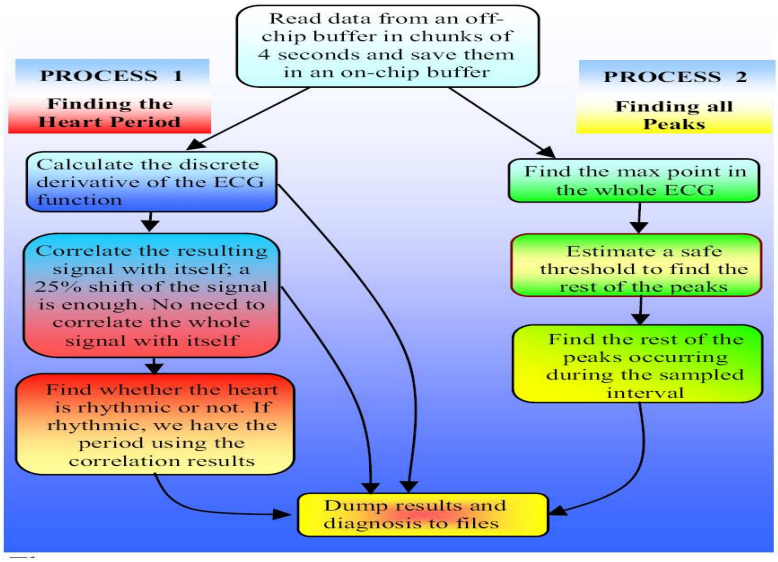


Figure 6.7: The Autocorrelation function-based methodology for ECG analysis.

the number of processing elements in the system. Alternatively, more leads can be processed by the same processor core provided the real-time requirements are achieved.

6.7 MPSoC Architecture

In order to process filtered ECG data in real-time, we chose to deploy a parallel Multi-Processor System-on-Chip architecture. The key point of these systems is to break up functions into parallel operations, thus speeding up execution and allowing individual cores to run at a lower frequency with respect to traditional monolithic processor cores. Technology today allows the integration of tens of cores onto the same silicon die, and we therefore designed a parallel system with up to 13 masters and 16 slaves (6.8). Since we are targeting a platform of practical interest, we chose advanced industrial components [96]. The processing elements are multi-issue VLIW DSP cores from STMicroelectronics, featuring 32KB instruction and data caches. Processor speed can achieve 400 MHz, although 200 MHz can be preferred in more power-aware solutions. These cores leverage the flexibility of programmable cores and the computation efficiency of DSP cores. Each processor core has its own private memory (512KB each), which is accessible through the bus, and can access an on-chip shared memory (8KB are enough for this application) for storing computation results. Other relevant slave components are a semaphore slave, implement-

ing the test-and-set operation in hardware and used for synchronization purposes by the processors or for accessing critical sections, and an interrupt slave, which distributes interrupt signals to the processors. Interrupts to a certain processor are generated by writing to a specific location mapped to this slave core. The STBus interconnect from STMicroelectronics was instantiated as the system communication backbone. STBus can be instantiated both: as a shared bus or as a partial or full crossbar, thus allowing efficient interconnect design and providing flexible support for design space exploration. Bus frequency is 200 MHz. In our first implementation, we target a shared bus to reduce system complexity (6.8) and assess whether application requirements can already be met or not with this configuration. We then explore also a crossbar-based system, which is sketched in 6.9. The inherent increased parallelism exposed by a crossbar topology allows decreasing the contention on shared communication resources, thus reducing overall execution time. In our implementation, only the instantiation of a 3x6 crossbar was interesting for the experiments. We put a private memory on each branch of the crossbar, which can be accessed by the associated processor core or by a DMA engine for off-chip to on-chip data transfers. Finally, we have a critical component for system performance which is the memory controller. It allows efficient access to the external 64MB SDRAM off-chip memory. A DMA engine is embedded in the memory controller tile, featuring multiple programming channels. The controller tile has two ports on the system interconnect: one slave port for control and one master port for data transfers. The overall controller is optimized to perform long DMA-driven data transfers. Embedding the DMA engine in the controller has the additional benefit of minimizing overall bus traffic with respect to traditional standalone solutions. Our implementation is particularly suitable for I/O intensive applications such as the one we are targeting in this work. In the above description, we have reported the worst case system configurations. In fact, fewer cores can be easily instantiated if needed. In contrast, this architectural template is very scalable and allows for further future increase in the number of processors. This will allow to run in real time even more accurate ECG analyses for the highest sampling frequency available in sensors (10,000Hz, and 15 leads, for instance), since this platform is able to provide scalable computational power. The entire system has been simulated by means of the MPSIM simulation environment [96], which provides for cycle-accurate functional simulation of complete MPSoCs at a maximum simulation speed of about 200Kcycles/second (running on a P4 at 3.5GHz). The simulator provides also a power characterization framework leveraging 0.13μm technology-homogeneous industrial power models from STMicroelectronics [97], [98]. We believe that for life-critical applications such as ECG real-time analysis, it is

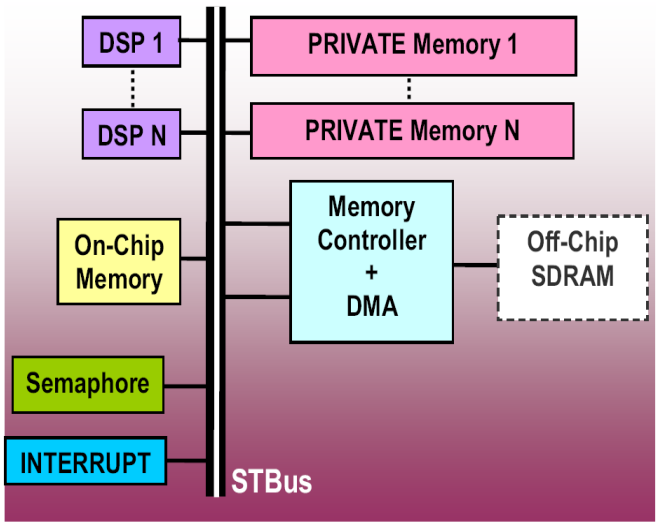


Figure 6.8: Single bus architecture with STBus interconnect.

important to conduct low-level accurate simulations in order to perfectly understand system level behaviour and have a predictable system with minimum degrees of uncertainty.

Each processor core programs the DMA engine to periodically transfer input data chunks onto their private on-chip memories. Moved data typically corresponds to 4 seconds of data acquisition at the sensors: 10KB at 1000Hz sampling frequency, transferred on average in 319279 clock cycles (DMA programming plus actual data transfer) on a shared bus with 12 processors. The consumed bus bandwidth is about 6MBytes/sec, which is negligible for an STBus interconnect, whose maximum theoretical bandwidth with 1 wait state memories exceeds 400Mbyte/sec. Then each processor performs computation independently, and accesses its own private memory for cache line refills.

Different solutions can be explored, such as processing more leads onto the same processor, thus impacting the final execution time. Output data, amounting to 64 bytes, are written to the on-chip shared memory, but their contribution to the consumed bus bandwidth is negligible. In principle, when the shared memory is filled beyond a certain level, its content can be swapped by the DMA engine to the off-chip SDRAM, where the history of 8 hours of computation can be stored. Data can also be remotely transmitted via a telemedicine link.

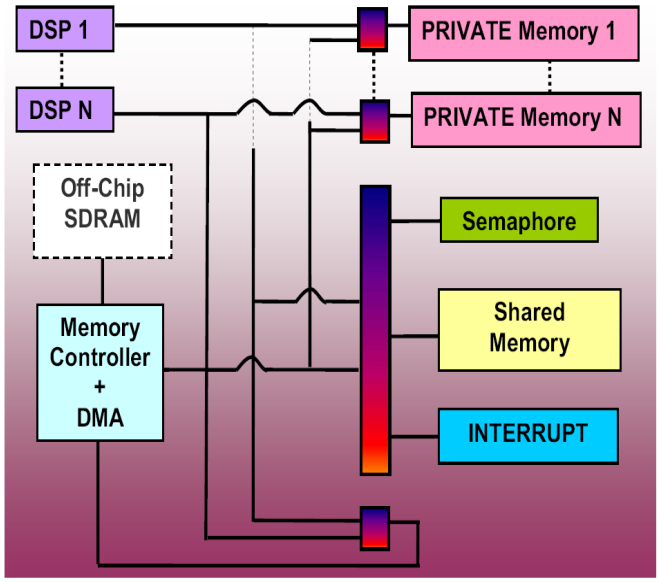


Figure 6.9: Crossbar architecture with STBus interconnect. Low-bandwidth slaves have been grouped to the same crossbar branch (partial crossbar concept).

6.8 Experimental Results

The first analysis was done to profile the execution of the code and to determine the best coding solution in terms of energy, execution time, and precision. Furthermore, we have explored the design space searching for the best platform configuration for the 12-lead ECG data analysis. Alternative system configurations have been devised for different levels of residual battery lifetime, trading off power with accuracy.

6.8.1 Floating Point vs Fixed Point Code

We ran two different code implementations: (a) one using floating point variables and (b) one using fixed point integers [130] with an exponent of 22. 6.10 shows the results for the two different code implementations from time (execution time) and energy (relative) points of view. The ST220 processor core runs at 200MHz. We have performed the analysis for 3, 6 and 12 leads; furthermore we process each lead on a separate core.

We found that the precision of the results obtained with fixed point code, by using 64 bit integer data types representation, almost matches the results obtained with floating point code for a large number of input data traces. On the contrary, the time needed to process data, and also the energy required, decreases up to 5 times. This is mainly due to the fact that, like many commercial DSPs, our processor cores do not have a dedicated floating point unit.

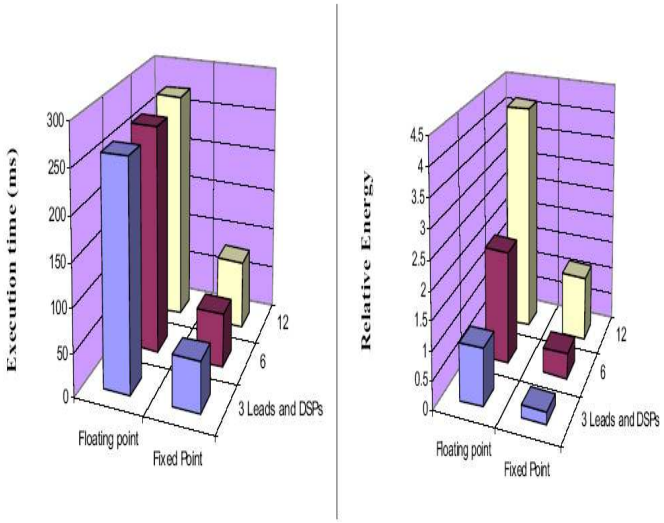


Figure 6.10: Comparison between different code implementations for the analysis of the 3-lead, 6-lead and 12-lead ECG. Data analysis for each lead is computed on a separate processor core. Sampling frequency of input data was 250Hz. System operating frequency was 200 MHz.

Therefore, floating point computations are emulated by means of a C software library linked at compile time. 6.10 also shows that even with 12 concurrent processors, the bus is not saturated, since we observe negligible effects on the stretching of task execution times. In contrast, adding more processors determines a linear increase in energy dissipation.

6.8.2 Comparison between Processor Cores

We then compared the performance of an ARM7TDMI with the ST220 DSP core, in order to assess the relative performance of the chosen VLIW DSP core with respect to a reference and popular architecture for general purpose computing, when put at work to process the computation kernel of our specific application. In order to have a safe comparison, we set similar dimensions of the cache memory (32KB) for the two solutions, and we run two simulations for the processing of one ECG-Lead at 250Hz sampling frequency. We count execution cycles to make up for the different clock frequencies. We adopt this single-core solution, since our first aim is to investigate the computation efficiency of the two cores for our specific biomedical application, and de-emphasize system level interaction effects such as synchronization mismatches or contention latency for bus access. In 6.11, we can observe that the ST220 DSP proves more effective both in execution time and energy consumption, as expected. In de-

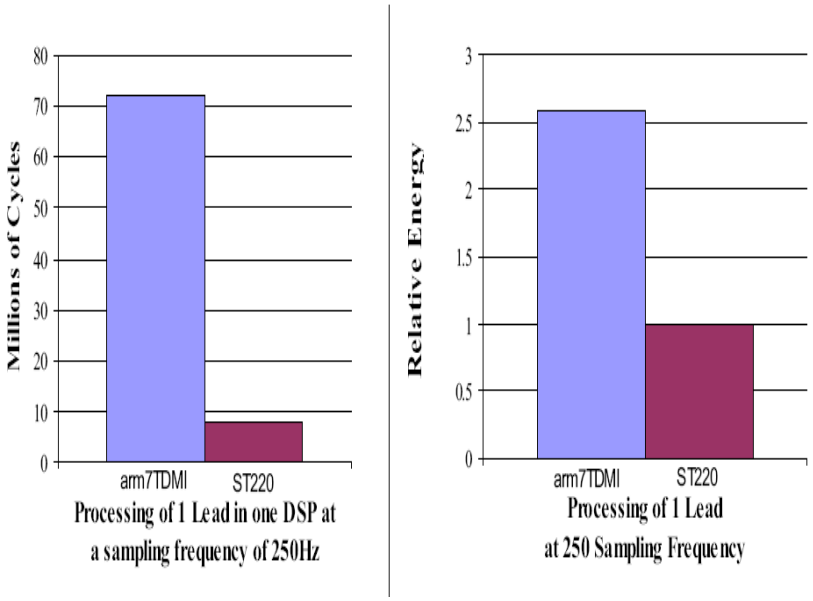


Figure 6.11: Comparing ARM7TDMI with ST200 DSP performances, when processing 1 Lead at 250Hz sampling frequency.

tail, the ARM core is 9 times slower than the ST220 in terms of execution time, and it consumes more than twice the energy incurred by the DSP. These results can be explained based on three considerations:

- The ST220 has better software development tools, which result in a smaller executable code. The size of the executable code for the ARM is 1.7 times larger than that of the ST220.
- The ST220 is a VLIW DSP core, therefore it is able to theoretically achieve the maximum performance of 4 instructions per cycle (i.e., 1 bundle).
- A metric which is related to both previous considerations is the static instructions per-cycle, which depends on the compiler efficiency and on the multi-pipeline execution path of the ST220. For our application, this metric turns out to be 2.9 instructions-per-bundle for ST220.

6.8.3 Allocation of Computation Resources

Based on previous findings (Sections 7.1 and 7.2), we will adopt a HW/SW architecture consisting of the ST220 DSP core and a fixed point coding implementation of the algorithm for the experiments that follow. The ST220 will be operated at its typical frequency of 400MHz, while the rest of the system will run at 200 MHz. We now want to optimally configure the system to satisfy the application requirements at the minimum hardware cost. We there-

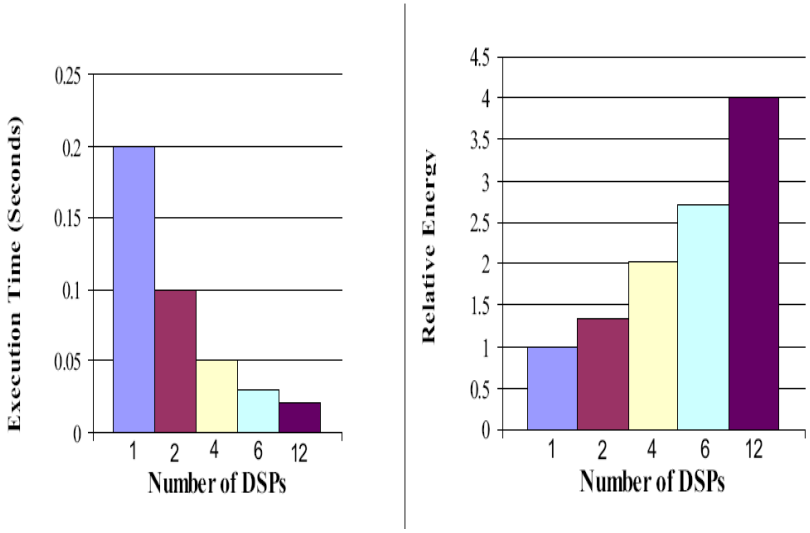


Figure 6.12: Execution Time and relative energy of the system with an increasing number of DSPs and input data sampled at 250Hz sampling frequency. System interconnect is a shared bus.

fore measure the execution time and the energy dissipation for an increasing number of DSP cores in order to find the optimal configuration of the system. Since commercially available ECG solutions target sampling frequencies ranging from 250 to 1000Hz, we performed the exploration for these two extreme cases for the 12-lead ECG signal. We analyze a chunk of 4secs of input data, which provides a reasonable margin for safe detection of heartbeat disorders. Figure 12 shows that if we increase the number of processors, the execution time scales almost linearly, at least up to 6 processors. After that, we observe diminishing returns in increasing system parallelism. Since the real-time requirement of 4 seconds for the overall computation is largely met, we conclude that in the range of interest (up to 6 processors) second order effects typical of multi-processor systems (e.g., bus contention reducing the offered bandwidth to the processor cores with respect to the requested one) are negligible. A single shared bus and even a single processor core are well suited for this case. However, this does not mean that the amount of data moved across the bus is negligible. This data is, however, read by the processor cores throughout the entire execution time, thus absorbing only a small portion of the bus bandwidth. In this regime, bus performance is still additive, i.e. the bus delivers a bandwidth which equals the sum of the bandwidth requirements of the processor cores.

Moreover, the good scalability of the application is also due to memory controller performance. In fact, at the beginning of the computation each processor loads processing data from the off-chip to the on-chip memory, hence,

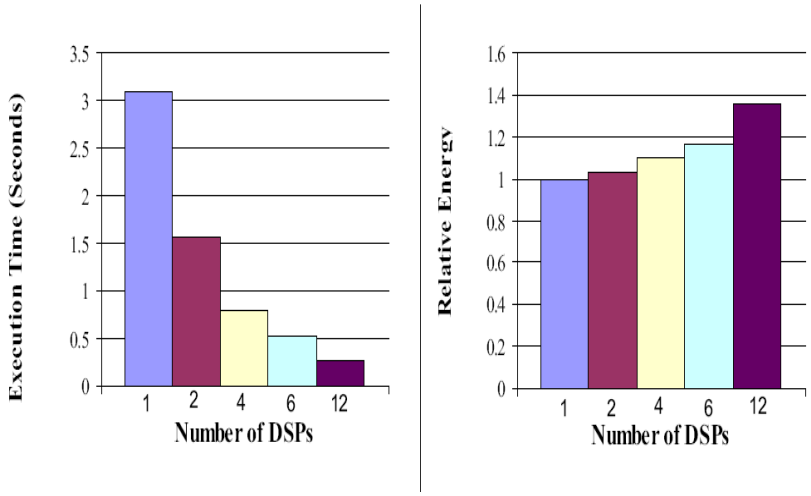


Figure 6.13: Execution Time and relative energy of the system with an increasing number of DSPs and input data sampled at 1000Hz sampling frequency. System interconnect is a shared bus.

requiring peak memory controller bandwidth. The architecture of the memory controller proves capable of providing the required bandwidth in an additive fashion. By looking at the 1000Hz plot (6.13), we observe that for the single processor case, the time it takes for a DSP to process 12 leads increases by more than 15 times with respect to the 250Hz case. Energy has increased as well by 90%. We still have about 1 second margin before the deadline (4 seconds), which is enough to perform additional analysis of the results of the individual lead-computations and converge to a diagnosis based on computed heartbeat parameters. In case a larger margin is needed, the increased workload can be effectively tackled by activating a larger number of processor cores. This comes at smoother energy degradation than the 250 Hz case, as showed in 6.13 (for the 1KHz sampling frequency). The larger number of energy consuming cores is better amortized by the savings on application execution. Although even for the 1KHz case, 1 DSP already meets the real-time requirements, the inherent parallelism of our architecture is useful in many senses. Firstly, when the margin to the deadline is too tight to run a complex diagnosis algorithm, the execution time can be reduced by using more processors. Secondly, working with a large number of processors allows sustaining higher sampling frequencies than 1KHz and more complex algorithms for high accuracy analysis. Thirdly, more processors can help save power, since instead of running one processor at full-speed, we may want to run more processors at reduced speeds thus cutting down on overall system energy.

An overview of the performance and energy overhead that is incurred

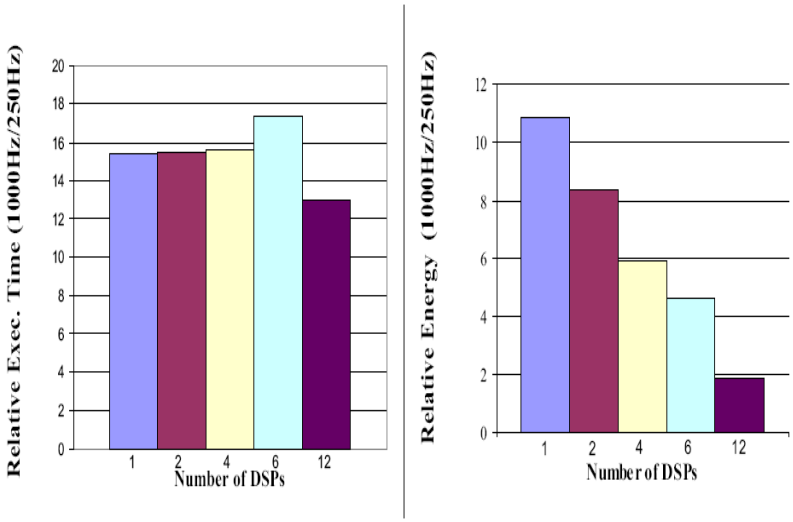


Figure 6.14: Relative Execution Time and Energy Ratios between the 1000Hz and the 250Hz sampling frequency experiments.

when moving from 250Hz to 1000KHz sampling frequencies of input data is reported in 6.14. Interestingly, the performance plot shows a constant 15x increase in computation time up to 4 processors. In the 6 processor case, the larger amount of data which needs to be transferred on the bus by each processor (due to data over-sampling) determines an increase of bus access times and therefore a longer execution time. As we push system parallelism to the limit, we observe (see the 12 DSPs case) that the computation workload is fully parallelized, and a huge but unique peak bandwidth is requested to the bus. Moving from 1 DSP to 12 DSPs, we move from 12 null contention bandwidth peaks to a single, heavy contention peak. This traffic profile shapes the execution time ratio curve as showed in 6.14. The energy-ratios plot confirms that the overhead for introducing more processors is worth in the 1000Hz case, while is not fully justified for the 250Hz case due to the different computation complexities to be tackled.

6.8.4 HW/SW Optimization for Aggressive Scalability

We are interested in assessing the achievable upper bound in system performance. This paves the way for further improvements of the biomedical algorithm, and it supports the use of the high data acquisition capabilities of the state-of-the-art biomedical sensors (i.e. higher sampling frequencies). In order to push our HW/SW design to suit more accurate analysis while respecting the real-time constraint, we look at how we can push both: the specific-application algorithm (SW) and the HW architecture while considering the high medical

demands of correctness and accuracy of results at the service level (medical service). To have higher accuracy and be able to diagnose arrhythmias like the R-on-T phenomena [126] and other medical cases, we found that the biomedical analyses necessitate higher sampling frequencies as input. The need for analysis at higher frequencies delivers the reality that: not only do we need to look at HW issues, but we also have to look at the algorithm parameters. In previous experiments, we used a 4-second input chunk to leave a safety margin for the input signals, and we used the number of Lags (L) variable to compensate for the data chunk size. We found that in the case of higher frequencies we can change some parameters so that the input data chunk can be optimized while still keeping good service (medical) level results. The solution is that we restrict the analysis chunk-size of our biomedical algorithm to 3.5 seconds (instead of 4 seconds), which also effects the number of multiplications that are needed. From the HW viewpoint, we simulated a 12 processor system performing the 12-lead ECG analysis with increasing sampling frequencies to determine the threshold value beyond which the system does not converge to a solution in real-time. We found that the limit for the input sampling frequency to be 2200Hz (maximum). We verified that in this operating condition, system performance is communication-limited, i.e. the shared bus architecture is not able to keep up with the increase in communication bandwidth requirements any more. Therefore, we face the need to push the hardware as the algorithm was pushed to the maximum. By further performing hardware optimization, we were able to replace the shared bus with a full crossbar, and observed that 12 leads could be processed then in slightly more than 1 second, i.e. well below the 3.5 seconds deadline. Such an optimized HW/SW architecture was proved to work in real-time up to a sampling frequency of 4000Hz (Fig. 6.15).

In this condition, the system turns out to be computation-dominated, hence the communication architecture is not the bottleneck. The flexibility of our system interconnect allows to achieve the same performance with less hardware resources. In fact, a partial crossbar design was experimented, consisting of grouping low bandwidth cores on the same crossbar branch. We observed that performance with the partial crossbar closely matches that of a full-crossbar (less than 2% average difference) but with almost 3 times less hardware resources. We found the optimal crossbar configuration (5x5 instead of 13x13) by accurate characterization of shared bus performance. On a shared bus, we increased the number of processors and observed when the execution time started deviating as an effect of bus contention. With up to 4 cores connected to the same communication resource, this latter is still able to work in an additive regime. Hence, it is not necessary to use full crossbars, but partial crossbars can be equally effective with less hardware resources.

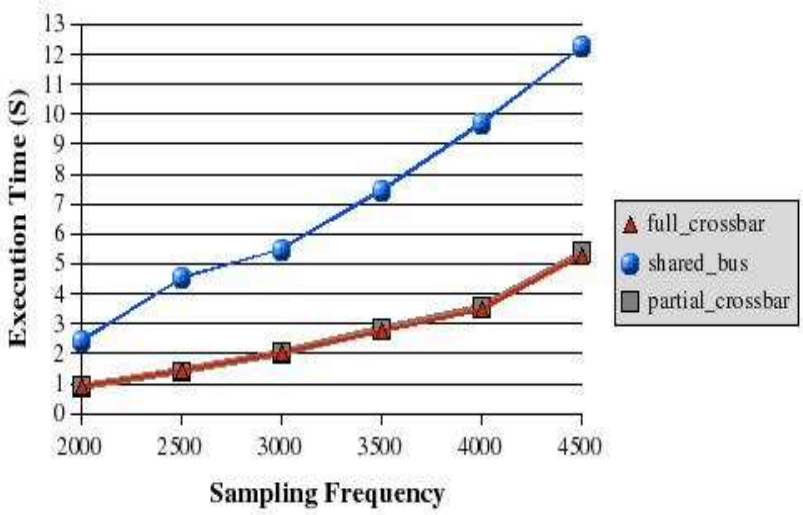


Figure 6.15: Critical sampling Frequencies for 3 architectures: (1) shared bus, (2) full crossbar, and (3) partial crossbar.

6.8.5 Conclusion and Future Work

We present an application-specific MPSoC architecture for real-time ECG analysis, which paves the way for novel healthcare delivery scenarios (e.g., mobility) and for accurate diagnosis of heart-related diseases in real-time. Although a single DSP architecture proves capable of meeting the real-time requirements of our biomedical applications for lower than the maximum (10KHz) that state-of-the-art biomedical-sensors can deliver, the inherent parallelism we provide prevents the architecture from being the bottleneck for further advances in the field of ECG analysis. Our biochip solution can support the increasing sampling frequencies of biomedical sensors and the increased computation efficiency of analysis algorithms optimized for accuracy. We propose a case of such algorithms, leveraging auto-correlation function as a better performing alternative to the traditional and commonly-used Pan-Tompkins algorithm. An in-depth comparison of these algorithms goes beyond the scope of this paper, and is left for future work. The hardware architecture was built based on industrial components, and its performance upper bounds were clearly identified. The optimized HW/SW platform proves capable of dealing with up to 4000Hz sampling frequencies, when system performance becomes computation-limited.

Chapter 7

Conclusions

One of the most important problems for design space exploration of state-of-the-art SoCs is the availability of a flexible and accurate simulation platform. To this purpose, the development of MP-ARM, a multi-processor SoC simulation tool, has been extensively discussed throughout this work. It is able to simulate a scalable number of ARM or STLX cores interconnected to each other by means of an AMBA-compliant or STBus communication architecture. A parallel RTOS has been ported onto the platform, providing the system software support to run highly parallel applications.

This platform offers large potentials for research purposes. As an example, the performance of arbitration algorithms for AMBA buses has been investigated. We show that they perform differently under different communication patterns. In particular, slot reservation outperforms other schemes in presence of tasks characterized by different workloads and that have to synchronize to each other during execution. On the contrary, round robin exhibits very good performance both for the case of independent tasks and of pipelined tasks. We point out the inability of TDMA to efficiently accommodate interactive inter-node handshakes and the need for a matching between hardware and software to maximize system performance.

We have then explored programming paradigms for parallel multimedia applications on MPSoCs. Our analysis points out that the trade-offs spanned by MPSoC platforms can be very different from those of traditional parallel systems, and provide some design guidelines to discriminate between message passing and shared memory programming paradigms in relevant subspaces of the software space. We show that message passing is not only a forward-looking solution for highly integrated network-on-chip based MPSoCs, but can be applied also to small scale on-chip multiprocessors depending on

application characteristics and on the availability of hardware extensions to efficiently support messaging. Looking forward to scalability issues and system optimization, we envision a hybrid approach to MPSoC development, where systems are composed of several clusters of shared memory nodes which communicate among them in a message passing-like fashion. Such a design paradigm would possibly exploit the advantages of both architectural templates, resulting in a power- and performance-optimized system. The task of splitting the computational workload among the clusters becomes a key issue for the programmer, and it is an open and interesting research area.

Finally we present an application-specific MPSoC architecture for real-time ECG analysis, which paves the way for novel healthcare delivery scenarios (e.g., mobility) and for accurate diagnosis of heart-related diseases in real-time. Although a single DSP architecture proves capable of meeting the real-time requirements of our biomedical applications for lower than the maximum (10KHz) that state-of-the-art biomedical-sensors can deliver, the inherent parallelism we provide prevents the architecture from being the bottleneck for further advances in the field of ECG analysis. Our biochip solution can support the increasing sampling frequencies of biomedical sensors and the increased computation efficiency of analysis algorithms optimized for accuracy. We propose a case of such algorithms, leveraging auto-correlation function as a better performing alternative to the traditional and commonly-used Pan-Tompkins algorithm. An in-depth comparison of these algorithms goes beyond the scope of this paper, and is left for future work. The hardware architecture was built based on industrial components, and its performance upper bounds were clearly identified. The optimized HW/SW platform proves capable of dealing with up to 4000Hz sampling frequencies, when system performance becomes computation-limited.

Bibliography

[1] F. Balarin et al."Hardware-Software Co-Design of Embedded Systems: The Polis Approach," *Kluwer Academic Press*, 2007.

[2] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *Int. Journal of Computer Simulation*, Vol. 4, pp. 155 182, April 1994.

[3] J. Rowson "Hardware-Software Co-simulation ," *DAC-31: 31st De-sign Automation Conference*, pp. 439-440, June 1994.

[4] A. Ghosh et al. "A Hardware-Software Co-Simulator for Embedded System Design and Debugging," *ASPDAC 95: Asia South Pacific Design Automation Conference*, pp. 155-164, January 2000.

[5] K. Hines, G. Borriello "Dynamic Communication Models in Embedded System Co-Simulation," *DAC-34: ACM/IEEE Design Au-tomation Conference*, pp. 395 400, June 1998.

[6] Synopsys, Inc., "Eaglei"
http://www.synopsys.com/products

[7] Mentor Graphics Inc., "Seamless CVE"
http://www.mentor.org/seamless

[8] CoWare, Inc., "N2C"
http://www.coware.com/cowareN2C.html

[9] G. De Micheli "Hardware Synthesis from C/C++ Models," *DATE 99: Design Automation and Test in Europe*, pp. 382 383, March 1999.

[10] Synopsys, Inc., "SystemC, Version 2.0"
http://www.systemc.org

[11] L. Semeria, A. Ghosh "Methodology for Hardware/Software Co-verification in C/C++ ," *ASPDAC 00: Asia South Pacific Design Au-tomation Conference*, Vol. 14, No. 2, pp. 16-25, April June 1997.

[12] C. Liem, F. Nacabal, C. Valderrama, P. Paulin, A. Jerraya ",” *System-on-Chip Co-Simulation and Compilation, IEEE Design and Test*, Vol. 14, No. 2, pp. 16-25, April June 1997.

[13] J. Liu, M. Lajolo, A. Sangiovanni-Vincentelli “Software Timing Analysis Using HW/SW Co-Simulation and Instruction Set Sim-ulator,” *International Workshop on Hardware-Software Codesign*, International Workshop on Hardware-Software Codesign, pp. 65 69, March 1998.

[14] P. Gerin, S. Yoo, G. Nicolescu, A. Jerraya “Scalable and Flexible Co-Simulation of SoC Designs with Heterogeneous Multi-Processor Target Architectures,” *Asian-Pacific Design Automation Conference*, pp. 63 68, January 2001.

[15] K. Lahiri, A. Raghunathan, G. Lakshminarayana, S. Dey “Communication Architecture Tuners: a Methodology for the Design of High-Performance Communication Architectures for System-on-Chips,” *DAC-37: Design Automation Conference*, pp. 513-518, June 2000.

[16] GNU Project Web server
http://www.gnu.org/software

[17] Data Display Debugger, “GNU Project”
http://www.gnu.org/software/ddd

[18] Philips Nexperia Media Processor
http://www.semiconductors.philips.com/platforms/nexperia/media_processing

[19] Mapletree Networks Access Processor
http://www.mapletree.com/products/vop_tech.cfm

[20] Intel IXS1000 media signal processor
http://www.intel.com/design/network/products/wan/vop/ixs1000.htm

[21] Magnusson P.S., Christensson M., Eskilson J., Forsgren D., Hallberg G., Hogberg J., Larsson F., Moestedt A., Werner B. “Simics: A full system simulation platform,” *IEEE Trans. on Computer*, Volume: 35 Issue: 2 , Feb 2002.

[22] Rosenblum M., Herrod S.A., Witchel E., Gupta A. “Complete computer system simulation: the SimOS approach,” *IEEE Parallel and Distributed Technology: Systems and Applications*, Volume: 3 Issue: 4 , Winter 1995.

[23] Hughes C.J., Pai V.S., Ranganathan P., Adve S.V. “Rsim: simulating shared-memory multiprocessors with ILP processors,” *IEEE Trans. on Computer*, Volume: 35 Issue: 2 , Feb 2002.

[24] Mentor Graphics Seamless, "Hardware/Software Co-Verification"
http://www.mentor.com/seamless/products.html

[25] CoWare Inc. "N2C"
http://www.coware.com/cowareN2C.html

[26] Van Rompaey K., Verkest D., Bolsens I., De Man H. "CoWare-a design environment for heterogeneous hardware/software systems," *Design Automation Conference, 1996, with EURO-VHDL '96 and Exhibition*, Proceedings EURO-DAC '96, 16-20 Sep 1996.

[27] Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Steve Huss-Lederman, Mark D. Hill, James R. Larus, and David A. Wood. "Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture," *Workshop on Performance Analysis and Its Impact on Design (PAID)*, June 1997.

[28] Babak Falsafi and David A. Wood "Modeling Cost/Performance of a Parallel Computer Simulator," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, January 1997.

[29] Lahiri K., Raghunathan A., Lakshminarayana G., Dey S. "Communication architecture tuners: a methodology for the design of high-performance communication architectures for system-on-chips," *Design Automation Conference, 2000*, Proceedings 2000. 37th ,2000 Page(s): 513 -518.

[30] Lahiri K., Raghunathan A., Lakshminarayana G. "LOTTERYBUS: a new high-performance communication architecture for system-on-chip designs," *Design Automation Conference, 2001*, Proceedings, 2001.

[31] Anjo K., Okamura A., Kajiwara T., Mizushima N., Omori M., Kuroda Y. "NECoBus: a high-end SOC bus with a portable and low-latency wrapper-based interface mechanism," *Custom Integrated Circuits Conference, 2002*, Proceedings of the IEEE 2002 , 2002, Page(s): 315 -318.

[32] Kyeong Keol Ryu, Eung Shin, Mooney V.J. "A comparison of five different multiprocessor SoC bus architectures," *Digital Systems, Design, 2001*, 2001 Page(s): 202 -209.

[33] Synopsys Inc., "SystemC Version 2.0"
http://www.systemc.org

[34] G. De Micheli "Hardware Synthesis from C/C++ Models," *Design Automation and Test in Europe*, Mar. 1999.

[35] M. Dales, "SWARM software Arm"
http://www.dcs.gla.ac.uk/ michael/phd/swarm.html

[36] ARM, "AMBA bus"
http://www.arm.com/armtech.nsf/html/AMBA?OpenDocument&style=AMBA

[37] J.Cong "An Interconnect-Centric Design Flow for Nanometer Technologies," *Int. Symp. VLSI Technology, Systems, and Applications*, pages 54–57, June 1999.

[38] uClinux
www.uclinux.org

[39] Oka and Suzuoki, "Designing and Programming the Emotion Engine," *IEEE Micro*, vol. 19-6, no. 8, pp. 20–28, 1999.

[40] D. Wingard, "MicroNetwork-Based Integration for SOCs," in *DAC*,Las Vegas, June 2001.

[41] Virtual Socket Interface Alliance,
http://www.vsi.org

[42] CoreConnect Bus Architecture,
http://www.chips.ibm.com/products/coreconnect

[43] ARM, "AMBA Specification Overview"
http://www.arm.com./Pro+Peripherals/AMBA

[44] W. Peterson, "Design Philosophy of the Wishbone SoC Architecture," *Silicore Corporation,1999*, http://www.silicore.net/wishbone.htm.

[45] K. Lahiri, A. Raghunathan and G. Lakshminarayana, "LOTTERYBUS: A New High-Performance Communication Architecture for Systems-on-Chip Design," in *Proceedings of DAC 2001*, pp. 15–20, Las Vegas (USA), June 2001.

[46] L. Benini and G. De Micheli, *Networks on Chip: a new SoC Paradigm*. Computer, 35(I):70-78, January 2002.

[47] D. Bertozzi, L. Benini, A. Bogliolo, F. Menichelli, G. Olivieri, "MPARM: Exploring the Multi-Processor SoC Design Space with SystemC," submitted to *Journal of VLSI Signal Processing*, January 2003.

[48] D.Wingard and A. Kurosawa"Integration Architecture for System-on-a-Chip Design", *Proceedings of IEEE 1998 Custom Integrated Circuits Conference*, pp.85-88, May 1998

[49] K. Lahiri, A. Raghunathan and S. Dey "Efficient Exploration of the SoC Communication Architecture Design Space," *IEEE-ACM Intern. Conference on Computer Aided Design 2000*, pp.424-430, San Jose (USA), 2000.

[50] M.E. Kreuz, L. Carro, A. Zeferino and A.A. Susin"Communication Architectures for Systems-on-Chip," *14th Symposium on Integrated Circuits and Systems Design*, pp.14-19, Brazil, 2001.

[51] K.K.Ryu, E. Shin and V.J. Mooney"A Comparison of Five Different Multiprocessor SoC Bus Architectures," *Euromicro Symposium on Digital Systems Design*, pp.202-209, Poland, 2001.

[52] J. Liang,S.Swaminathan and R. Tessier "aSOC: A Scalable, Single-Chip Communication Architecture," *Int. Conference on Parallel Architectures and Compilation Techniques*, pp.37-46, Philadelphia (USA), 2000.

[53] W. Cesario, A. Baghdadi, L. Gauthier and D. Lyonnard"Component-Based Design Approach for Multicore SoCs," *Proceedings of DAC 2002*, pp.789-794, New Orleans (USA), 2002.

[54] R.A. Bergamaschi and W.R. Lee"Designing Systems-on-Chip Using Cores," *Proceedings of DAC 2000*, pp.420-425, Los Angeles (USA), 2000.

[55] A. Colin and I. Puaut"Worst-case execution time analysis of the RTEMS real-time operating system," *Euromicro Conference on Real-Time Systems*, pp.191-198, Netherlands, 2001.

[56] E.S. Shin, V.J.Mooney III and G.F. Riley "Round-Robin Arbiter Design and Generation," *Int. Symposium on System Synthesis*, pp.243-248, Kyoto (Japan), 2002.

[57] A. Jerraya and W. Wolf,"Multiprocessor Systems-on-Chips," *Morgan Kaufmann, Elsevier*, 2005.

[58] G. Declerck, "A look into the future of nanoelectronics," *IEEE Symposium on VLSI Technology*, pp. 6-10, 2005.

[59] W. Weber, J. Rabaey, E. Aarts,(Eds.), *Ambient Intelligence*. Springer, 2005.

[60] D. Culler, J. Singh,A. Gupta, "Parallel Computer Architecture: A Hardware/Software Approach", Morgan Kaufmann Publishers, 1999.

[61] L. Hennessy, D. Patterson, "Computer Architecture – A Quantitative Approach", Morgan Kaufmann Publishers, 3rd edition, 2003.

[62] S. Hand, A. Baghdadi, M. Bonacio, S. Chae, and A. Jerraya. An efficient scalable and flexible data transfer architectures for multiprocessor SoC with massive distributed memory. In *Proc. 41 Dac*, pages 250–255, 2004.

[63] F.Gilbert, M. Thul, and N. When. Communication centric architectures for turbo-decoding on embedded multiprocessors. In *Proc. Date*, pages 356–351, 2003.

[64] H. A. et al. A 160mW, 80nA Standby, MPEG-4 Audiovisual LSI 16Mb Embedded DRAM and a 5 GOPS Adaptive Post Filter. In *IEEE int. solid-state circuits conference*, pages 62–63, 2003.

[65] M. Rutten, J. van Eijndhoven, E. Pol, E. Jaspers, P. van der Wolf, O. Gangwal, and A. Timmer. Eclipse: heterogeneous multiprocessor architecture for flexible media processing. In *Proc. int. parallel and distributed processing conf.*, pages 39–50, 2002.

[66] U. Ramachandran, M. Solomon, and M. Vernon. Hardware support for interprocess communication. *IEEE trans. parallel and distributed systems*, pages 318–329, Jul. 1990.

[67] M. Banekazemi, R. Govindaraju, R. Blackmore, and D. Panda. MP-LAPI: An efficient implementation of MPI for IBM RS/6000 SP systems. *IEEE trans. parallel and distributed systems*, pages 1081–1093, Oct. 2001.

[68] W. Lee, W. Dally, S. Keckler, N. Carter, and A. Chang. An efficient protected message interface. *IEEE Computer*, pages 68–75, Mar. 1998.

[69] N.E. Crosbie, M. Kandemir, I. Kolcu, J. Ramanujam, and A. Choudhary. Strategies for Improving Data Locality in Embedded Applications. *Proc. of the 15th Int. Conf. on VLSI Design*, 2002.

[70] M.M. Strout, L. Carter and J. Ferrante. Rescheduling for Locality in Sparse Matrix Computations. *Lecture Notes in Computer Science*, p.137, 2001.

[71] M. Kandemir. Two-Dimensional Data Locality: Definition, Abstraction, and Application. *Int. Conf. on Computer Aided Design*, pages 275–278, 2005.

[72] G. Byrd and M. Flynn. Producer-Consumer Communication in Distributed Shared Memory Multiprocessors. *Proc. IEEE*, pages 456–466, Mar. 1999.

[73] K.Tachikawa, "Requirements and Strategies for Semiconductor Technologies for Mobile Communication Terminals," *Electron Devices Meeting*, pp.1.2.1-1.2.6, 2003.

[74] D. Pham et al., "The design and implementation of a first-generation CELL processor," in *Proceedings of ISSCC*, February 2005.

[75] ARM Semiconductor, "ARM11 MPCore Multiprocessor", http://arm.convergencepromotions.com/catalog/753.htm

[76] Philips Semiconductor, "Philips Nexperia Platform", www.semiconductors.philips.com/products/nexperia/home

[77] STMicroelectronics Semiconductor, "Nomadik Platform", www.st.com/stonline/prodpres/dedicate/proc/proc.htm

[78] Texas Instrument Semiconductor, "OMAP5910 Platform", http://focus.ti.com/docs/prod/folders/print/omap5910.html

[79] MPCore Multiprocessors Family, www.arm.com/products/CPUs/families/MPCoreMultiprocessors.html.

[80] Intel Semiconductor, "IXP2850 Network Processor", Available at http://www.intel.com.

[81] B. Ackland et al., "A Single Chip, 1.6 Billion, 16-b MAC/s Multiprocessor DSP," *IEEE Journal of Solid State Circuits*, Vol. 35, No. 3, March 2000.

[82] C. Lin, L. Snyder, "A Comparison of Programming Models for Shared Memory Multiprocessors," *International Conference on Parallel Processing*, pages 163–170, 1990.

[83] T. A. Ngo and L. Snyder, "On the influence of programming models on shared memory computer performance," *International Conference on Scalable and High Performance Computing*, pages 284–291, 1992.

[84] T.J. LeBlanc, E.P. Markatos, "Shared memory vs. message passing in shared-memory multiprocessors," *Symposium on Parallel and Distributed Processing*, pages 254–263, Dec. 1992.

[85] A.C. Klaiber, H.M. Levy, "A Comparison of Message Passing and Shared Memory Architectures for Data Parallel Programs," *ISCA'94: International Symposium on Computer Architecture*, pages 94–105, 1994.

[86] S. Chandra, J. R. Larus, A. Rogers, "Where is Time Spent in Message-Passing and Shared-Memory Programs?" *ASPLOS'94: International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 61-73, 1994.

[87] S. Karlsson and M. Brorsson. "A comparative characterization of communication patterns in applications using MPI and shared memory on an IBM SPI," *International.Workshop on Communication, Architecture, and Applications for Network-Based Parallel Computing*, pages 189–201, 1998.

[88] H. Shan, J. P. Singh. "A comparison of MPI, SHMEM and cache-coherent shared address space programming models on the SGI Origin2000," *International. Conference on Supercomputing*, pages 329-338, 1999.

[89] H. Shan, J.P. Singh, L. Oliker, R. Biswas, "Message passing vs. shared address space on a cluster of SMPs," *International Parallel and Distributed Processing Symposium*, April 2001.

[90] D. Altilar, Y. Paker, "Minimum Overhead Data Partitioning Algorithms for Parallel Video Processing," in *12th Int. Conf. on Domain Decomposition Methods*, 2001.

[91] S. Bakshi, D.D. Gajski, "Hardware/Software Partitioning and Pipelining," in *ACM/IEEE DAC*, pages 713–716, 1997.

[92] W. Liu, V.K. Prasanna, "Utilizing the Power of High-Performance Computing", in *IEEE Signal Processing Magazine*, pages 85–100, Sep. 1998.

[93] Joao Paulo Kitajima, Denilson Barbosa, Wagner Meira Jr, "Parallelizing MPEG Video Encoding using Multiprocessors", in *SIBGRAPI*, pages 215-222, Sep. 1999.

[94] M. Stemm and R. H. Katz, "Measuring and reducing energy consumption of network interfaces in hand-held devices," *IEICE Transactions on Communications*, vol. E80-B, no. 8, pages 1125–31, 1997.

[95] V. Raghunathan, C. Schurgers, S. Park, and M. Srivastava, "Energy aware wireless microsensor networks," in *IEEE Signal Processing Magazine*, pages 40–50, Mar. 2002.

[96] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, R. Zafalon, "Analyzing On-Chip Communication in a MPSoC Environment", *Design and Test in Europe Conference (DATE)*, pages 752-757, Feb. 2004.

[97] M. Loghi, M .Poncino, L. Benini, "Cycle-Accurate Power Analysis for Multiprocessor Systems-on-a-Chip", in *GLSVLSI04: Great Lake Symposium on VLSI*, pages 401-406, Apr. 2004.

[98] Bona, A.; Zaccaria, V.; Zafalon, R., "System level power modeling and simulation of high-end industrial network-on-chip", in *Design and Test in Europe Conference (DATE)*, pages 318-323, Feb. 2004.

[99] Byrd, G.T.; Flynn, M.J., "Producer-consumer communication in distributed shared memory multiprocessors", in *Proceedings of the IEEE*, Volume 87, pages 456-466, March 1999.

[100] F.Poletti, P.Marchal, D.Atienza, L.Benini, F.Catthoor, J. M. Mendias., "An Integrated Hardware/Software Approach For Run-Time Scratchpad Management", in *Proceedings of the (DAC)*, Volume 2, pages 238-243, July 2004.

[101] Poletti F., Poggiali A., Marchal P., "Flexible hardware/software support for message passing on a distributed shared memory architecture", in *Proceedings of the (DATE)*, Volume 2, pages 736-741, March 2004.

[102] MPI-2 standard
http://www-unix.mcs.anl.gov/mpi/mpi-standard/mpi-report-2.0/mpi2-report.htm

[103] P. Stenström, "A Survey of Cache Coherence Schemes for Multiprocessors," *IEEE Computer*, Vol. 23, No. 6, June 1990, pages 12–24.

[104] M. Tomasevic, V. M. Milutinovic, "Hardware Approaches to Cache Coherence in Shared-Memory Multiprocessors," *IEEE Micro*, Vol. 14, No. 5–6, pages 52–59, October/December 1994.

[105] I. Tartalja, V. M. Milutinovic, "Classifying Software-Based Cache Coherence Solutions," *IEEE Software*, Vol. 14, No. 3, pages 90–101, March 1997.

[106] A. Moshovos, B. Falsafi, A. Choudhary, "JETTY: Filtering Snoops for Reduced Energy Consumption in SMP Servers", *HPCA'01* January 2001, pp. 85-97.

[107] C. Saldanha and M. Lipasti, "Power Efficient Cache Coherence", *High Performance Memory Systems*, Springer-Verlag, 2003, pages 63–78.

[108] M. Ekman, F. Dahlgren, P. Stenstrom, "Evaluation of Snoop-Energy Reduction Techniques for Chip-Multiprocessors," *ISCA'02*, May 2002.

[109] M. Ekman, F. Dahgren, P. Stenstrom, "TLB and Snoop Energy-Reduction Using Virtual Caches in Low-Power Chip-Multiprocessors," *ISLPED'02*, August 2002, pages 243–246.

[110] Ruggiero M., Guerri A., Bertozzi D., Poletti F., Milano M., "Communication-Aware Allocation and Scheduling Framework for Stream-Oriented Multi-Processor Systems-on-Chip", in *Proceedings of the (DATE)*, Volume 1, pages 3-9, March 2006.

[111] P. Banerjee, J. Chandy, M. Gupta, J. Holm, A. Lain, D. Palermo, S. Ramaswamy, and E. Su. Overview of the PARADIGM Compiler for

Distributed Memory Message-Passing Multicomputers. *IEEE Computer*, pages 37–37, Mar. 1995.

[112] M. Gupta, E. Schonberg, and S. H. A Unified Framework for Optimizing Communication in Data-Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, pages 689–704, Jul. 1996.

[113] Fuster V. "Epidemic of Cardiovascular Disease and Stroke: The Three Main Challenges," *Circulation*, pag 1132-1137, Vol. 99, Issue 9, March 1999.

[114] Heart and Stroke Foundation of Canada: The Changing Face of Heart Disease and Stroke in Canada 2000
Annal report (1999)

[115] Chan, C., Han, J., Ramjeet, D. "LabVIEWTM Design of a Vectorcardiograph and 12-Lead ECG Monitor," *Final Year Project for the Bachelor of Science Degree in the University of Manitoba*, March 2003.

[116] Ambu, Inc. biomedical devices company
www.ambuusa.com

[117] Harland, C., Clark, T., Prance, R. "Electric Potential Probes- New Directions in the remote sensing of the human body," *Measurement Science and Technology*, Vol. 13, (2002) 163-169.

[118] Harland, C., Clark, T., Prance, R. "High resolution ambulatory electrocardiographic monitoring using wrist-mounted electric potential sensors," *Measurement Science and Technology*, Vol. 14 (2003) 923-928.

[119] Malmivuo, J., Plonsey, R. "Bioelectromagnetism: Principles and Applications of Bioelectric and Biomagnetic Fields," *Oxford University Press*, 1995.

[120] Chevrollier, N., Golmie, N. "On the Use of Wireless Network Technologies in Healthcare Environments," *Proceedings of the fifth IEEE workshop on Applications and Services in Wireless Networks*, ASWN2005, June 2005, 147-152.

[121] Lo, B., Thiemjarus, S., King, R., Yang, G. "Body Sensor Network-A Wireless Sensor Platform for Pervasive Healthcare Monitoring," *Adjunct Proceedings of the 3rd International Conference on Pervasive Computing*, PERVASIVE'05, May 2005, 77-80.

[122] Association of Cardiac Technology in Victoria-ACTIV
http://www.activinc.org.au

[123] Code Blue-Wireless Sensor Networks for Medical Care
http://www.eecs.harvard.edu/ mdw/proj/codeblue

[124] BIOPAC Systems Inc.
http://biopac.com/

[125] Company-Bosch, E., Hartmann, E. "ECG Front-End Design is Simplified with MicroConverter," *Journal of Analog Dialogue*, Vol. 37, November 2003.

[126] Aaron Segal: EKG tutorial, "EMT-P (1997)"
http://www.drsegal.com/medstud/ecg

[127] Pan, J. and Tompkins, W. "A Real-Time QRS Detection Algorithm," *IEEE Transactions on Biomedical Engineering*, Vol. BME-32, No. 3, March 1985.

[128] PhysioBank, physiologic signal archives, for biomedical research
http://www.physionet.org/physiobank/database/ptbdb

[129] MIT-BIH arrhythmia database- Tape directory and format specification "Document BMEC TR00," Mass. Inst. Tech., Cambridge, 1980.

[130] ARM DAI 0033A Note 33
Fixed Point Arithmetic on the ARM, September 1996.