ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

FACOLTÀ DI INGEGNERIA

DOTTORATO DI RICERCA IN INGEGNERIA ELETTRONICA,
TELECOMUNICAZIONI E TECNOLOGIE DELL'INFORMAZIONE - CICLO XXV

SETTORE CONCORSUALE DI AFFERENZA: 09/E3
SETTORE SCIENTIFICO DISCIPLINARE: ING-INF/01

USE OF SHARED MEMORY IN THE CONTEXT OF EMBEDDED MULTI-CORE PROCESSORS: EXPLORATION OF THE TECHNOLOGY AND ITS LIMITS

PRESENTATA DA:
PAOLO BURGIO

COORDINATORE DOTTORATO
ALESSANDRO VANELLI-CORALLI

RELATORI
CHIAR.MO PROF. LUCA BENINI
CHIAR.MO PROF. PHILIPPE COUSSY

ESAME FINALE ANNO 2013/2014

Use of shared memory in the context of embedded multi-core processor: exploration of the technology and its limits

Paolo Burgio

Department of Electrical, Electronic
and Information Engineering "Guglielmo Marconi"
Universitá degli Studi di Bologna - Italy

and

Lab-STICC
Université de Bretagne-Sud - France

A thesis submitted for the degree of

PhilosophiæDoctor (PhD)

December 2013

Abstract

Nell'ultimo decennio, i sistemi integrati (*embedded systems*) hanno sposato le architetture *many-core*. Questo è stato necessario per continuare a soddisfare la richiesta di performance e di bassi consumi da parte del mercato, in particolare da quando smartphone e tablet sono entrati nella vita di tutti i giorni.

Per utilizzare al meglio questa tecnologia, spesso si assume che lo spazio di memoria disponibile sia totalmente condiviso fra i processori. Ciò è di grande aiuto sia perché semplifica il lavoro del programmatore, sia durante il design della piattaforma e del runtime stessi. Tuttavia, il paradigma a memoria condivisa è nato sui sistemi *symmetric multi-processors (SMP)*, e portarlo sulle moderne architetture embedded può essere problematico. Qui, per ragioni di dimensioni e di consumo di potenza, si tende a rimpiazzare le data cache con memorie *scratchpad*, che vanno gestite esplicitamente dal software, e quindi dal programmatore.

La situazione si complica ulteriormente se si considera il fatto che le moderne applicazioni devono essere *parallelizzati*, prima di poter essere eseguite sulle centinaia/migliaia di processori disponibili nella piattaforma. Per supportare la programmazione parallela, sono stati proposti diversi linguaggi, orientati a diversi tipi di piattaforme e paradigmi di programmazione. Tipicamente, questi linguaggi lavorano ad un alto livello di astrazione, pertanto abbisognano di un supporto sotto forma di libreria di *runtime*, che chiaramente ha un costo, un *overhead* che impatta negativamente sulle performance. Minimizzare questo costo è fondamentale. In questa dissertazione, esploro l'applicabilità del modello a memoria condivisa alle moderne architetture *many-core*, con particolare attenzione alla programmabilità. In particolare mi concentrerò su OpenMP, che è lo standard *de facto* per programmare i sistemi a memoria condivisa. In una prima parte della tesi vengono analizzati i servizi di base (sincronizzazione, allocazione della memoria) che si deve fornire a un livello più basso dello stack tecnologico, per consentire lo sviluppo di codice parallelo su un *many-core* a memoria condivisa, e in particolare il loro costo, e infine vengono proposte alcune soluzioni per renderli più snelli. In seguito ci si sposta a un livello più alto d'astrazione, e viene mostrato come un runtime di supporto al programming model OpenMP debba essere implementato in modo da supportare efficientemente più livelli di parallelismo, e un paradigma a parallelismo irregolare, a *tasking*, in un sistema *shared-memory*.

Nella seconda parte della tesi, il focus si sposta su un altro trend per il design dei sistemi integrati, cioè, la possibilità di includere acceleratori hardware nella piattaforma, che quindi viene detta *eterogenea*. Viene introdotto un template per un'architettura eterogenea *many-core*, dove acceleratori e processori comunicano tramite la memoria condivisa. Fissato lo schema di comunicazione, si può definire uno scheletro per un generico acceleratore (HWPU) che verrà specializzato a seconda dell'applicazione, e si può definire uno scheletro per una piattaforma con sia acceleratori che processori.

Una volta che il protocollo di comunicazione e la piattaforma sono definiti, si può sviluppato uno stack software e un insieme di *tool* per svilupparla, e per consentire alle applicazioni di sfruttare gli acceleratori hardware. Tutti i dettagli di basso livello, non necessari allo sviluppo dell'applicazione, sono "nascosti" in un runtime, e il programmatore interagisce con esso tramite annotazioni OpenMP. In questa dissertazione vengono presentati due approcci ortogonali: da un lato (*top-down*), si cerca di semplificare il lavoro del progettista della piattaforma, includendo i vari tool necessari allo sviluppo in un unico *toolflow*; dall'altro (*bottom-up*), si forniscono agli sviluppatori di software gli strumenti per sfruttare gli acceleratori hardware presenti nella piattaforma, senza appesantire il proprio codice.

Complimenti per l'ottimo acquisto

Acknowledgements

Premessa: visto che mi son rotto di inglese e francese, i ringraziamenti ve li beccate in italiano (vedi http://translate.google.com).

Stavolta, di scrivere un papiro come per la specialistica, non se ne parla neppure.

Per due principali motivi: uno, perché l'ho già fatto (e quindi andiamo in "maniera incrementale" per dirla alla ingegnere), e due perché alla fine della fiera, c'è solo una persona senza la quale questi 5 anni, con tutte le cose che son successe, mi sarebbero veramente passati sopra come un rullo compressore: me.

Ciononostante, alcune persone mi sono state vicine, magari anche solo per pochissimo, o magari mi hanno semplicemente – e involontariamente – detto la parola giusta al momento giusto: quello che mi serviva. Senza nulla volere alle altre, sto giro sono loro che voglio ringraziare.

Paraculatamente, i primi sono Luca, Philippe e Andrea (in ordine – immagino – di pubblicazioni). Grazie Luca: come ripetevo sempre, lavorare nella "serie A" (o almeno quella che credo sia la serie A), paga sempre e cmq, e lavorare per te è super-stimolante (anche se ha dei ritmi "un tantinello" elevati...). Grazie Philippe per avermi riempito di fiducia e responsabilità, l'ultimo passo che mi mancava per diventare quello che volevo (e che sono contento di) essere. Grazie Andrea per avermi sopportato, per esserti fatto sopportare, per le notti a casa tua, mia o in lab (spesso assieme alla "brutta persona" – grazie anche a lui) a lavorare, a suonare, a mangiare come dei maiali o anche solo al parco a demolire il Jesus Christ Superstar e i Queen. E concorderai di ringraziare i reviewer di CASES, che hanno accettato il Vertical Stealing e le sue menate matematiche frutto delle nostre menti perverse e di 30 ore da svegli. Se non fosse passato, credo che ci saremmo

suicidati o scannati a vicenda, dopo “una notte da leoni”. E per quella notte, ringrazio *sentitamente* i manutentori di distributori di merendine del terzo piano di Ing. per averli riempiti due giorni prima. Fra gli altri ragazzi del lab, in particolare io e il mio fegato ricordiamo con affetto Criscian. E poi il cast di *Pasiones Multicore*, e Luis Ceze, chiaramente.

Paraculate #2 – Grazie alla mia famiglia per avermi sopportato e per aver soprattutto sopportato la mia lontananza, soprattutto tu mamma so quanto ci tieni a vedermi e ad avermi vicino, ma questo non è stato quasi mai possibile. E purtroppo non posso garantirti che la situazione migliorerà in futuro. Mi spiace, avreste dovuto farmi nascere in un’altra nazione, o una 20na di anni prima...

Grazie – con scuse annesse – ai ragazzi di Ravenna. Con voi sono stato ancor meno presente che con la mia famiglia: vi basti sapere che mi mancate sempre e cmq, e non sto scherzando. E bona di giocare ad “Amici miei”, siete un po’ grandicelli! (o no?)

Grazie a Caps (coglione), Ele, la Lugli, e la Eli. Voi in particolare mi siete stati – gli unici – vicino quando ho toccato VERAMENTE il fondo. Sulemani.

E il più grosso **grazie** va all’*équipe* di Lorient: Moira, Lucie, Dominique, Maria, Steph, Vasco, Juli, Béréngere, Mariana&Tristan, Célie, Nelson, La Chiarina, Fabian, Salma, Nolo e Leandro (in ordine sparso, dovreste esserci tutti). Una volta, Célie mi disse che era troppo felice di avere un gruppo di amici così: io ribadisco il concetto. Nei prossimi anni, i km (e l’oceano) separeranno o riuniranno alcuni di noi.. sarà divertente vedere come va a finire!

Infine, grazie, a una persona che non ho neppure bisogno di nominare (tanto hai capito benissimo). E non serve neppure il motivo: non ci starebbe in questa pagina, e poi tu e io non abbiamo mai dovuto parlare, per dirci qualcosa.

Paolo

Contents

List of Figures

List of Tables

1

Introduction

1.1 Background: many-core shared memory architectures

In the last decade, computing systems entered the chip multiprocessor (CMP) era. Moore's law is still valid, but, as H. Sutter said, *"the free lunch is over"* (117): processors' manufacturers still follow the 40%-per-year increase of number of transistors *per* chip area, but they are not able anymore to scale performance by increasing the clock frequency and instruction throughput of single cores (24, 38, 45). Moreover, modern applications are increasing in complexity and must deliver high performance at low power budgets (4, 88, 118). A typical example is an audio/video stream from a web page, to be decoded in real-time on a mobile device such as a smartphone.

As a consequence, computing platforms adopted multi- and subsequently many-cores designs, where energy-efficient performance scaling is achieved by exploiting large-scale parallelism, rather than speeding up the single processing units (1, 15, 70, 83, 109). The trend involved both general purpose (55), high-performance (HPC) (64, 101), and embedded (15, 109) computing, and in this thesis we will mainly focus on the latter. In particular, the way communication between the different processing units is implemented greatly impacts the performance of platforms (12). A typical and effective solution is to implement it using shared memory banks (5, 12, 15, 109).

The abstraction of a shared memory was initially adopted in "traditional" symmetric multi-processors (SMP) systems, where processors are connected to a unique memory, and where multiple coherent level of caches are used to increase the locality of data to the core, fighting the so-called *memory wall* (20, 141). This paradigm is quite appealing for being adopted also in the embedded domain (12, 15), but it has some drawbacks.

In first instance, while caches are an effective solution for general-purpose systems, in the embedded domain things are quite different. Here, the area and energy budgets are significantly scaled down, and the big spatial and power overhead of a hierarchical cache system (35, 99, 100) can not anymore be tolerated. For this reasons, it is a common design choice (15, 70, 109) to partially or totally replace data caches with shared on-chip scratchpad memories (SPM): they are fast SRAMs, limited in size to meet the area budget, and supported by bigger and slower off-chip DRAMs, from which data is explicitly moved back and forth to increase locality to the processing cores, e.g., with DMA transfers. This means that the software – that is, embedded programmers – have now the burden of explicitly handling the shared memory (23, 62): this hinders *programmability*.

1.2 Programming shared-memory architectures

A second issue stems from the fact that many-core architectures have a tremendous potential in terms of parallelism and energy efficiency (Gops/Watt), but the task of turning it into actual performance is demanded at the software level, and at programmers' skills. This is a non-trivial task: in 2010, Blake et al. (23) highlighted that most of the existing desktop applications are not able to exploit more than a few cores at the same time. To this aim, several languages/extensions were proposed, that provide constructs (such as keywords or code annotations) to enable parallel code development at a high level of abstraction (44, 63, 96, 106). Typically, low-level services – such as thread and memory management, and orchestration of the data movements – are transparent to programmer, and implemented inside a runtime library, which in general-purpose systems leverages on the underlying operating system. In the embedded domain, replicating this software stack *as is* is not possible, due to its overhead. Since embedded systems have limited resources, typically runtime supports lie on a lightweight *micro-kernel*, or run directly on bare metal using the *board support package* (BSP).

This thesis starts from these issues, and investigates how a highly expressive parallel programming model can be efficiently supported on embedded many-cores, in such a way that parallelism and the shared-memory management are as much as possible transparent to programmers. Among the several possible programming languages (7, 63, 96), we will focus on OpenMP (106): it grew in the mid-90s out of the need to standardize the different vendor specific directives related to parallelism, and has subsequently emerged as the *de-facto* standard for programming shared memory SMPs. The great expressiveness of OpenMP lets the analyses and solutions shown in this work to be applicable to most of the existing programming language for embedded systems. OpenMP exposes a simple and lightweight interface for parallel programming, and preserving this simplicity of use is the primary goal of this work. However, due to the aforementioned issues, it is necessary to modify and somehow "complicate" the software stack, to handle parallelism and (explicit) memory management in such a way they they are transparent to programmers. For this reason, most of the the techniques proposed here are implemented ("hidden") in the runtime layer. Where it is not possible, few additional language constructs and extensions are proposed to expose them at the application level.

1.3 Programming heterogeneous architectures

The second part of the thesis focuses on heterogeneous architectures. Nowadays, power and energy efficiency are the primary concern for the embedded systems market, as portable devices such as smartphones and tablets went mainstream (4, 88, 118, 138). To cope with this issues, a widely adopted solution is to exploit architectural heterogeneity (27, 82, 115, 118) of the platforms. Designers are increasingly including hardware accelerators in many-core platforms, to implement key *kernels* with orders-of-magnitude of speedup and energy efficiency compared to software counterparts (36, 48, 66, 118). These *kernels* are implemented in ASIC/FPGA technologies, and the corresponding hardware blocks are coupled to general-purpose cores in a tighly or loosely manner.

The second part of the thesis explores how architectural heterogeneity impacts programmability and how it changes the process of designing many-core systems. Coupling one or more hardware accelerators and several general purpose cores poses three main challenges, namely *platform design*, *architectural scalability* and *programmability*. The former two mean providing system architects with architectural templates (82, 114), methodologies and tools (17, 30, 133) to support and automate the design process of the platform, while the latter means providing languages, compilers and runtime libraries for developing software that effectively exploits its acceleration opportunities. Especially, programmability is still an open issue, and in both industry and academia there is a general effort for doing research in this direction. To give an example, in the Khronos (127) consortium a set of key players of the market are pushing towards a common set of standard APIs for programming heterogeneous computer vision and image processing systems, called OpenVX (128).

Both from the software and platform design viewpoints, the process of integrating accelerators and many-cores is greatly simplified by clearly defining the communication protocol between them. The shared-memory communication mechanism is again beneficial, especially from a programmability perspective, because it reduces the data copies to the ones for increasing data locality. This happens, e.g., as opposite to GPU-based systems, where data must be explicitly moved to/from the accelerator's tile *in any case* before and after the actual computation (for instance using OpenCL (80)). This thesis explores how, assuming shared-memory communication, we can simplify the task of platform design and software development for heterogeneous many-core systems. Starting from a clearly defined "communication contract", an architectural template for heterogeneous many-core platforms is shown, and a design flow and tools for supporting it are proposed. Then, an efficient runtime support for communication based on shared-memory is developed, and (existing or brand new) higher-level programming abstractions (such as the ones proposed by OpenMP (107) or OpenACC (105)) can be enriched and ported on top of it. OpenMP perfectly fits the shared-memory communication scheme, and is adopted also in this part.

1.4 Overview of the Thesis

This document covers four years of work. During this period, designs for embedded platforms significantly changed, following the market trends and technological advances, and so did the software design process. The generic baseline architecture targets modern embedded platforms, where small *clusters* of (up to 16) cores are replicated and connected through a Network-on-Chip to scale to many-cores. This is shown in chapter 2, which also gives a detailed overview of the OpenMP programming model.

The remaining four chapters are the main contributions of my thesis.

Chapter 3 analyzes the costs for supporting OpenMP (and, more in general, any parallel programming model) on many-core embedded systems featuring a shared-memory. Typically these platforms run a lightweight/reduced operating system (such as a Linux Vanilla kernel (89)) or a *micro-kernel* where multiple threads cooperately execute parallel tasks on the different processing units. OpenMP was originally designed for general purpose systems, where low-level services such as synchronization, memory allocation and multi-threading are provided by the operating system, and where resources (such as the size of stack, heap and caches) are less constrained than in the embedded world. When applied to embedded systems, "traditional" mechanisms for multi-threading and synchronization come with an overhead that in some cases is significant (97), harnessing performance and limiting their applicability to units of work that are *coarse* enough to amortize their cost (42). Thus, they must be redesigned for being effectively adopted also on these platform.

Chapter 4 explores different task allocation strategies on an architecture with partitioned, 3D-stacked shared-memory. Several workload distribution schemes are considered, and their applicability to the target platform is analyzed, in such a way they are applicable to fine-grained units of work, and scalable to tens and hundred of concurrent processors. Custom extensions to OpenMP are shown, to effectively expose them to the application level.

Chapter 5 studies how a runtime for supporting nested and irregular parallelism can be efficiently implemented on shared-memory many-cores. OpenMP was traditionally designed for general purpose systems expressing regular, loop-based parallelism, whereas modern applications also exploit more complex partitioning schemes, e.g., with a first level of coarse-grained parallel tasks, and then a second level of SIMD-like parallelism, for instance using parallel loops. It is therefore crucial to efficiently supporting this so-called *nested parallelism* on a shared-memory system. Moreover, modern applications are growing in complexity, and expose a form of parallelism more irregular and dynamically created at runtime than simple SIMD-like parallelism. This is often referred to as *tasking* or *work-queues*. Such a complex behavior is supported by the underlying runtime library, whose overhead greatly affects the minimum granularity of tasks that can be efficiently spawned on embedded platforms.

The last – and latest – part of my thesis focuses on *heterogeneous* platforms for embedded systems. This part is a cooperation between University of Bologna and

Université de Bretagne-Sud, under a joint PhD agreement. In chapter 6 an architecture is proposed where hardware accelerators are tightly-coupled to cores inside shared-memory clusters. As already pointed out, the three main challenges in such a system are *platform design*, *architectural scalability* and *programmability*, and this chapter shows how they can be tackled. Communication between cores and accelerators happens through tightly-coupled shared-memory banks, implementing the so called *zero-copy* scheme (5), and accelerators embodying it are called Hardware Processing Units (HWPUs). Hardware accelerators can become a nightmare to programmers, who are required to write *scalable*, *modular* and *portable* code (as good programming practice says) that uses them. As a first contribution, this chapter describes the design for a lightweight runtime layer to support cores-to-HWPUs *zero-copy* communication, and proposes a lightweight set of extensions to the OpenMP set of APIs to efficiently exploiting it from the application layer. The template for heterogeneous clusters and a design methodology for it are shown, and tools are developed to automate the design process. Subsequently, it is shown how heterogeneous clusters inherently suffer from scalability issues when attempting to insert several (tens of) accelerators in the design. An architectural variant is introduced to cope with this, where a so-called Data Pump module interfaces the HWPUs to the on-cluster shared memory, and provide support for efficiently programming them.

2

Target architecture and programming model

The purpose of this Chapter is to give an overview of the baseline architecture and the target programming model that are considered in this document. They are not intended to fully cover the topics, but rather will help the reader providing a minimal – yet exhaustive – background for reading my thesis. For more details, interested reader may refer to the specific references mentioned.

2.1 Shared-memory many-core clusters

The architecture considered in this work follows a well-known design trend of modern embedded many-core systems. In a *clustered* platform, processing cores are grouped into small sets (i.e., few tens), which are highly optimized for performance and throughput. Clusters are the "building blocks" of the architecture, which scales to many-core by replicating them and connecting them through a scalable medium such as a Network-on-Chip (14). In such a design each cluster has a separate clock frequency domain, thus implementing a GALS architecture (Globally Asynchronous Locally Synchronous). Notable examples are ST Microelectronics' STHORM/P2012 (15), Plurality Hypercore Architecture Line (HAL) (109), Kalray MPPA (70), Adapteva Epiphany IV (1) or even GP-GPUs such as NVIDIA Fermi (101). Figure 2.1 shows a platform with three clusters and an off-chip memory. The Figure also shows a typical design choice for such an architecture, the one of sharing the memory space, which is partitioned among

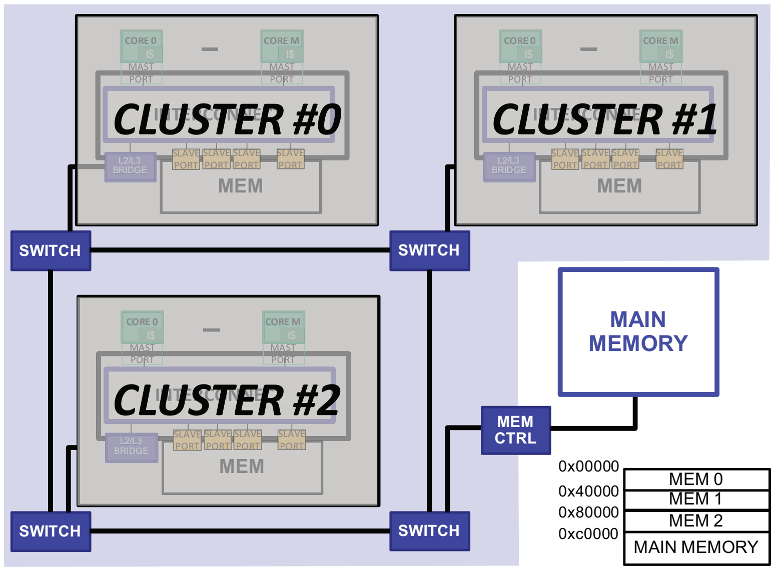


Figure 2.1: Multi-cluster shared-memory architecture

the different clusters (on-chip memories) and the bigger – and slower – off-chip L3 memory, resulting in a Partitioned Global Address Space (PGAS). Not shown in the Figure, a partitioned shared on-chip memory can also be featured, hosted on a separate cluster, or 3D-stacked on top of the chip. Every core can access every memory location in the system, experiencing different latencies and thus resulting in a NUMA hierarchy. Internally, clusters have aggressive throughput-oriented designs and memory systems. Figure 2.2 shows a possible template for the single cluster: it features 16 RISC cores, each of which has a private instruction cache. There is no data cache, but rather a multi-banked and multi-ported shared scratchpad memory, accessed through an on-cluster crossbar.

A Network Interface enables off-cluster data access, and a special shared bank featuring *test-and-set* programming is provided to support synchronization. The amount of on-cluster memory is limited (typically, hundreds of kilobytes to few megabytes), thus the full data set is initially stored in larger L2/L3 memories, and each cluster is equipped with a DMA engine to efficiently on/offload the mostly referenced data set, increasing its locality to the processing cores. Double buffering techniques are a typical solution that developers employ to hide this latency. As we will see in each chapter, the specific cluster is designed depending on the different goals considered time after time, and following different platform generations.

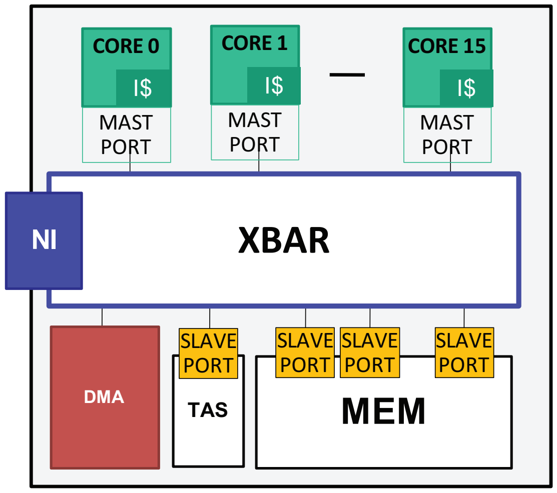


Figure 2.2: Single shared-memory cluster

A brief overview of some notable architectures follows.

2.1.1 ST Microelectronics' P2012/STHORM

Platform 2012 (15) is a low-power programmable many-core accelerator platform for embedded system, designed by ST Microelectronics. It is structured in clusters of cores, connected through a Globally Asynchronous Network-on-Chip (GANOC) and featuring a shared memory space between the cores. Clusters are internally synchronous, and the global architecture is GALS (Globally Asynchronous Locally Synchronous).

Figure 2.3 shows the internal structure of a single cluster. Each cluster features a Cluster Controller (CC) and a multi-core system called ENCORE made of 16 processing units. All cores are a proprietary 32-bit ISA, STxP70-V4, each of which features a floating point unit (FPx), private instruction caches, and no data caches.

Processors are interconnected through a low-latency high-bandwidth logarithmic interconnect, and communicate through a fast multi-banked, multi-ported *tightly-coupled data memory* (TCDM) of 256kB. The number of memory ports in the TCDM is equal to the number of banks to allow concurrent accesses to different banks. Conflict-free TCDM accesses have two-cycles latency.

The logarithmic interconnect is built as a parametric, fully combinational mesh-of-trees (MoT) interconnection network (see Figure 2.4). Data routing is based on address decoding: a first-stage checks if the requested address falls within the TCDM

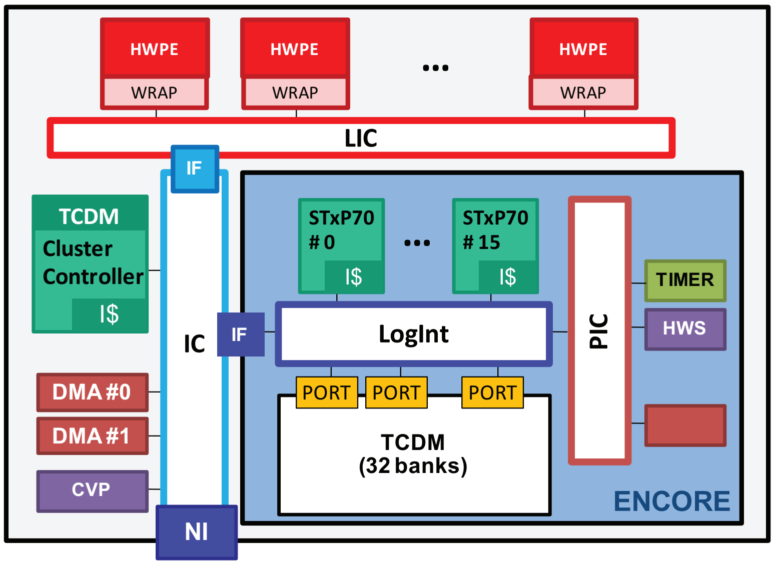


Figure 2.3: Simplified scheme of a P2012 configurable cluster

address range or has to be directed off-cluster. The interconnect provides fine-grained address interleaving on the memory banks to reduce banking conflicts in case of multiple accesses to logically contiguous data structures. The crossing latency is one clock cycle. If no bank conflicts arise, data routing is done in parallel for each core. In case of conflicting requests, a round-robin based scheduler coordinates accesses to memory banks in a fair manner. Banking conflicts result in higher latency, depending on the number of conflicting requests.

Each cluster is equipped with a Hardware Synchronizer (HWS) which provides low-level services such as semaphores, barriers, and event propagation support, two DMA engines, and a Clock Variability and Power (CVP) module. The cluster template can be enhanced with application specific hardware processing elements (HWPEs), to accelerate key functionalities in hardware. They are interconnected to the ENCORE with an asynchronous local interconnect (LIC).

Platform 2012 (and its first release, named STHORM, which features 4 homogeneous clusters for a total of 69 cores) comes with a software stack based on two programming models, namely a component-based Native Programming Model (NPM) and OpenCL-based (named CLAM – CL Above Many-Cores). OpenMP support is under development.

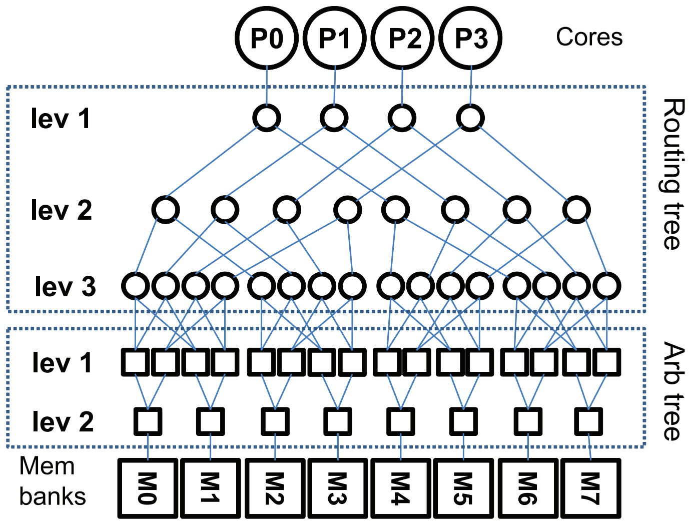


Figure 2.4: Mesh of trees 4x8

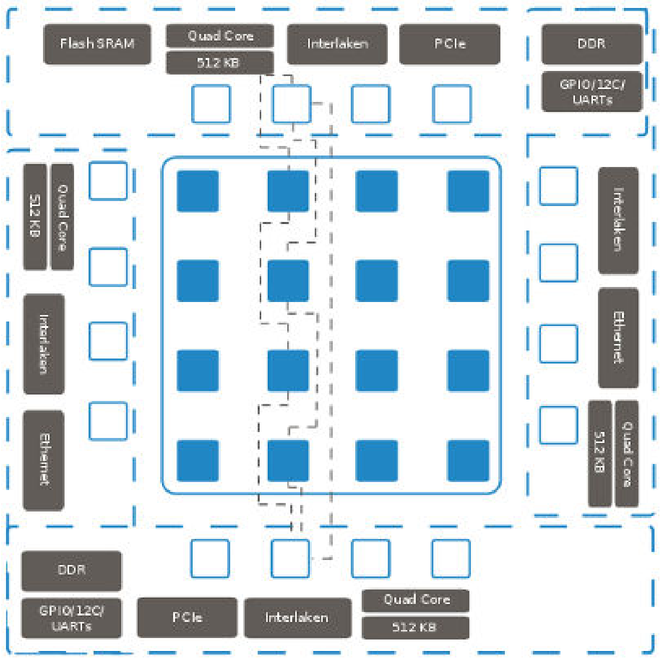


Figure 2.5: Kalray MPPA multicluster architecture

2.1.2 Kalray MPPA

Kalray Multi Purpose Processor Array (MPPA) (70) is a family of low-power many-core programmable processors for high-performance embedded systems. The first product, MPPA-256, embeds 256 general-purpose cores divided in 16 tightly-coupled clusters, connected through a Network-on-chip. It is shown in Figure 2.5. Each core is a proprietary (named K1) 32-bit processor, with its own instruction and data cache. Each cluster has a 2MB shared data memory, connected to processors through a full-crossbar. Clusters are and arranged in a 4×4 grid, and at its sides, four I/O clusters provide off-chip connectivity through PCI (North/South clusters) or Ethernet (West/East). Each I/O cluster has a four-cores processing units, and N/S clusters have a DDR controller to a 4GB external memory. The platform acts as an accelerator for an x86 host, connected *via* PCI to the north I/O cluster. Compute clusters run a lightweight operative system named NodeOS, while I/O clusters run an instance of RTEMS (102).

Two programming modes are provided, namely a dataflow programming and Posix-based programming. Dataflow programming is based on SigmaC, a C/C++-like programming model. Posix-based programming enables the usage of OpenMP (106) at the level of the single cluster. A channel-based support for inter-cluster communication is provided by the runtime library. OpenMP implementation is based on a proprietary compiler and runtime for the target K1 processor.

An integrated development environment (IDE) is provided, based on Eclipse (126), which provides debug and tracing capabilities. Using that, applications can be developed an deployed on the real board (if connected to the host), or with a simulator.

2.1.3 Plurality Hypercore Architecture Line

Plurality Hypercore (109) is an energy efficient general-purpose machine made of several RISC processors (from 16 up to 256). It is shown in Figure 2.6. Figure also shows the single processor structure, which is quite simple (e.g., has neither caches nor private memory, no branch speculation, etc..) for keeping the energy and area budgets low. The memory system (i.e., I/D caches, off-chip main memory) is fully shared, and processors access it through a high-performance logarithmic interconnect (see Figure 2.4. Processors share one or more Floating Point Units, and one or more shared hardware accelerators can be embedded in the design.

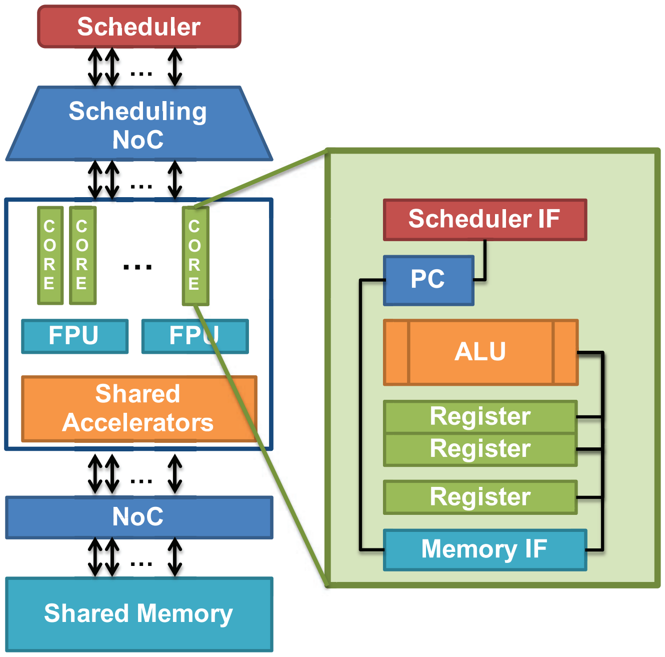


Figure 2.6: Plurality Hypercore Architecture Line (HAL), and single core

This platform can be programmed with a *task*-oriented programming model, where the so-called *"agents"* are specified with a proprietary language. Tasks are efficiently dispatched by a scheduler/synchronizer called Central Synchronizer Unit (CSU), which also ensures workload balancing. Dependencies between tasks are specified on a token-basis, and both task-parallelism (*regular* tasks) or data-parallelism (*duplicable* tasks) are supported.

2.2 Shared-memory programming models

In last decade, several parallel programming models were adopted for embedded systems. Probably the most famous is OpenCL (80), which attempts to provide a common standard for programming generic many-core accelerators, e.g., GPUs. However, in OpenCL the – hierarchical – memory space is explicitly non-shared (for instance, processing cores on the accelerator tile cannot access the global memory on the host side), thus data copies must be explicitly inserted, making the programming style cumbersome, and in some cases causing a complete rewrite of applications. Interesting is, OpenCL is well-known to provide portability if code, but not "portability of performance", which is rather demanded at the specific implementation of the – necessary – runtime support.

Sequoia++ (51) provides C++ extensions to expose the memory hierarchy to programmer, who must manually orchestrate data transfer to achieve locality of data and, eventually, performance.

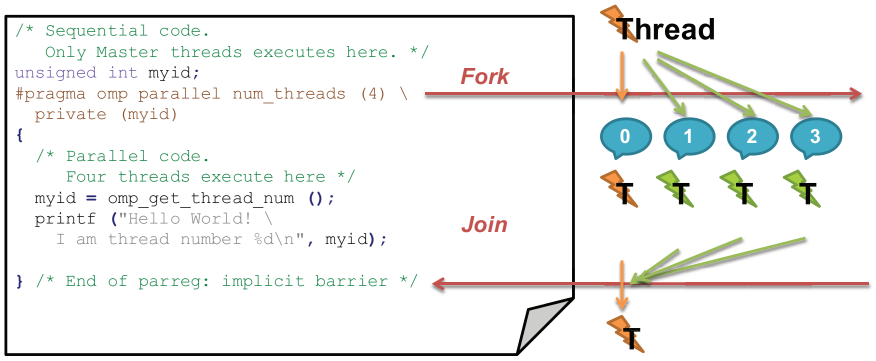
OpenMP (106) provides a lightweight set of APIs for C, C++ and Fortran to specify parallelization opportunities. It was initially developed for general purpose system, where the programmer is provided with the abstraction of a shared memory space, and where data locality is silently improved using (more levels of) caches. With OpenMP, the programmer works at a higher level of abstraction, with parallel threads (or tasks, since specifications 3.0), without worrying of the underlying memory management. This is provided *via* code transformations by the compiler, which works synergistically with the runtime library that provides thread creation and joining, synchronization and (transparent) memory consistency. In general purpose system, typically the OpenMP runtime leverages low-level services provided by the underlying Operative System, such as Pthreads (it's the case of the – reference – GCC-OpenMP implementation (52)) or custom libraries (such as OMPi with Psthreads (2), or BSC Mercurium with NANOS++ (123)). A careful and *ad-hoc* implementation of this runtime is paramount to achieving performance on the target –resource constrained – systems.

There is a whole "family" of shared memory-based programming languages that embody a *task-based* programming model, that is, they let the programmer explicitly partition an application onto parallel task, and provide mechanism for fork-join. Notable examples are Cilk (96), Intel Threading Building Blocks (63), or the (explicitly) queue-based Apple Grand Central Dispatch (7), as well as the aforementioned OpenMP since specifications 3.0 (dated 2009) or Plurality's proprietary language (109).

This work focuses on OpenMP because:

1. it's the *de-facto* standard for memory programming: most of the parallel programmers are already familiar with it, and several applications already exist written with it;

2. it's annotation-based: is lightweight set of APIs enables fast application porting, with minimal modifications to existing code;



```
/* Sequential code.
   Only Master threads executes here. */
unsigned int myid;
#pragma omp parallel num_threads (4) \
  private (myid)
{
  /* Parallel code.
     Four threads execute here */
  myid = omp_get_thread_num ();
  printf ("Hello World! \
    I am thread number %d\n", myid);

} /* End of parreg: implicit barrier */
```

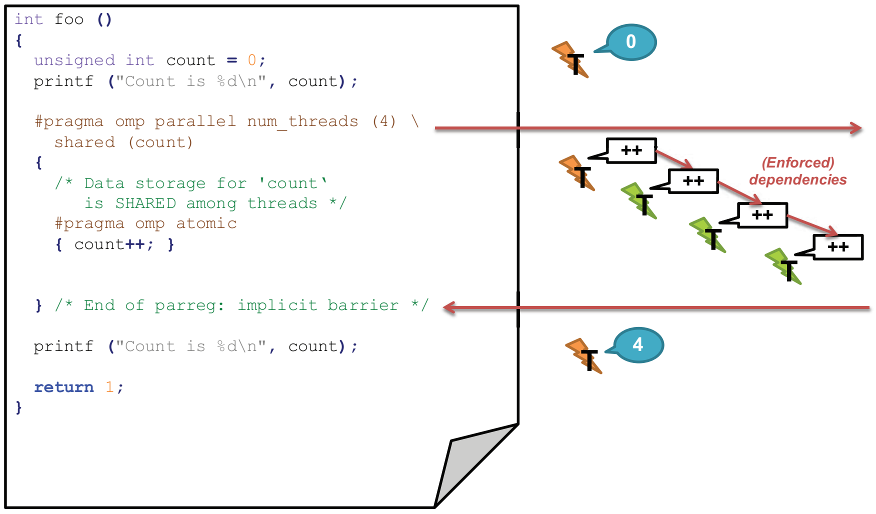
Figure 2.7: OpenMP parallel region

3. it provides a wide range of constructs for specifying most of the common parallel programming patterns: data parallelism (*static and dynamic loops*), task parallelism (*sections* and *tasks*), explicit synchronization (*barriers*), fork-join, *critical* sections, and so on. This makes all of the techniques and analysis that we will see in the thesis applicable to most of other programming languages.

2.3 The OpenMP programming model

OpenMP (106) is a set of compiler directives, environment variables, and runtime APIs to support thread-based parallelism on a shared-memory system.

Figure 2.7 shows an example of the OpenMP *fork-join* execution model. At startup, a *master* Thread (in orange) executes the program, and upon encountering a `parallel` construct, it spawns a team of *worker/slave* threads (in green), which in this example are explicitly requested to be four. The four threads execute in parallel the code enclosed within the so-called *parallel region*, and they are implicitly joined at its end, that is, each parallel region ends with an *implicit thread barrier*. Figure also shows the usage of `omp_get_thread_num ()` API, used to retrieve the unique ID assigned to a thread in the scope of a parallel region. OpenMP implicit and explicit barriers (enforced with the `pragma omp barrier` construct) are synchronization points for the threads, the only point of a program at which memory consistency is ensured. This implements the so-called *OpenMP relaxed memory model*. The `private` clause shown in the example specifies that the storage of the automatic variable `myid` is local to each executing thread, as opposite to the (default) `shared` clause which specifies a



```
int foo ()
{
  unsigned int count = 0;
  printf ("Count is %d\n", count);

  #pragma omp parallel num_threads (4) \
    shared (count)
  {
    /* Data storage for 'count'
       is SHARED among threads */
    #pragma omp atomic
    { count++; }


  } /* End of parreg: implicit barrier */

  printf ("Count is %d\n", count);

  return 1;
}
```

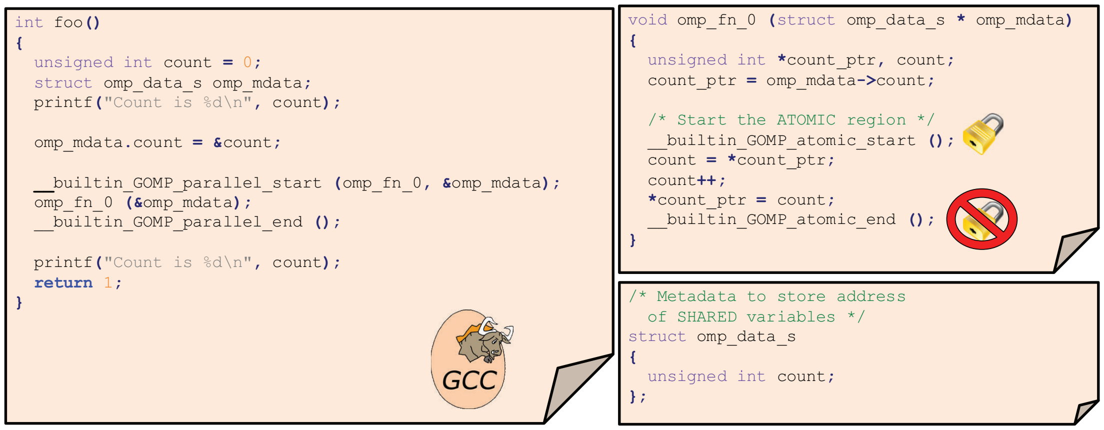
Figure 2.8: OpenMP `shared` variables and `atomic` clause

variable whose memory location will be shared among threads. In this case, OpenMP specifications state that the programmer is responsible for ensuring data consistency and avoiding data races for `shared` variables, e.g., using the `atomic` and `critical` constructs. This is shown in Figure 2.8.

To implement such complex behaviors, an OpenMP program is modified by the compiler, so to replace the pragmas with calls to the runtime implementation. Figure 2.9 shows how the code in Figure 2.8 is modified by the GCC-OpenMP compiler (GOMP) (52). As shown, the compiler inserts calls to the runtime APIs (which are implementation-specific – in this case they start with the prefix `GOMP_`) to handle respectively the parallel region and the atomic region. For example, it is responsibility of the `GOMP_atomic_start ()` and `GOMP_atomic_end ()` to ensure that only one thread at a time enters the portion of code annotated as `atomic`, by leveraging on low-level mechanism provided by the hardware architecture (e.g., atomic locks), or software libraries (e.g., mutexes).

2.3.1 Workload partitioning

Threads belonging to a parallel team execute the same (annotated) code. OpenMP provides several constructs for specifying workload partitioning. They are called **work-sharing** constructs. Figure 2.10 shows the `pragma omp sections` construct, which

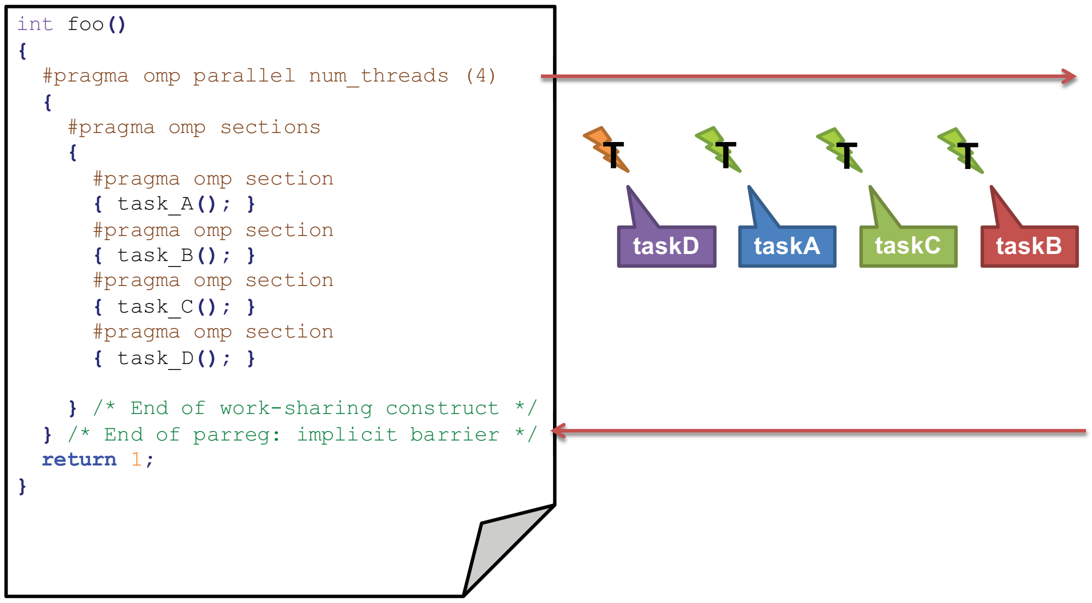


```
int foo()
{
  unsigned int count = 0;
  struct omp_data_s omp_mdata;
  printf("Count is %d\n", count);

  omp_mdata.count = &count;

  __builtin_GOMP_parallel_start (omp_fn_0, &omp_mdata);
  omp_fn_0 (&omp_mdata);
  __builtin_GOMP_parallel_end ();

  printf("Count is %d\n", count);
  return 1;
}
```

```
void omp_fn_0 (struct omp_data_s * omp_mdata)
{
  unsigned int *count_ptr, count;
  count_ptr = omp_mdata->count;

  /* Start the ATOMIC region */
  __builtin_GOMP_atomic_start ();
  count = *count_ptr;
  count++;
  *count_ptr = count;
  __builtin_GOMP_atomic_end ();
}
```

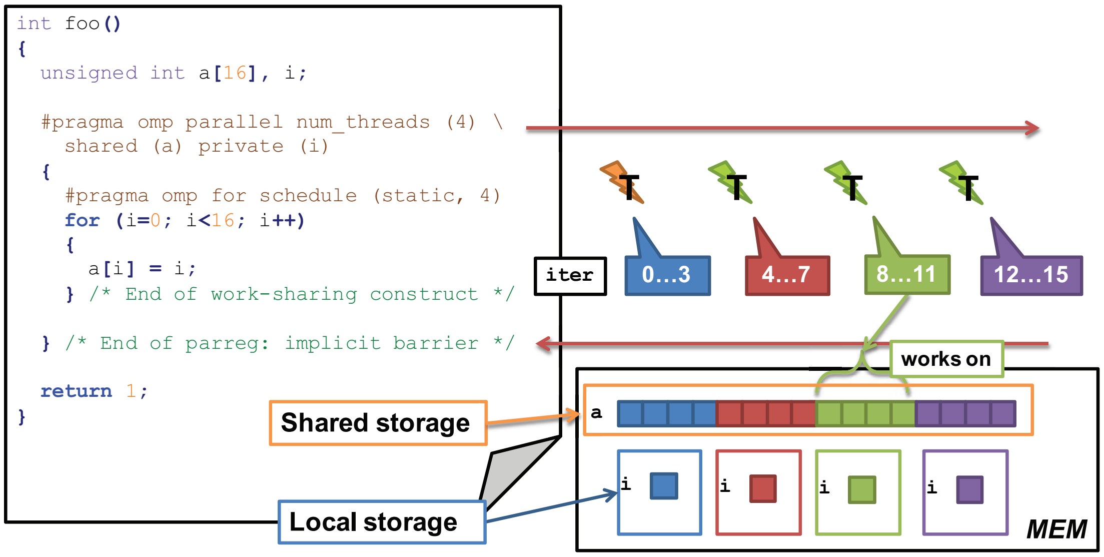
```
/* Metadata to store address
  of SHARED variables */
struct omp_data_s
{
  unsigned int count;
};
```

Figure 2.9: Modified GOMP code



```
int foo()
{
  #pragma omp parallel num_threads (4)
  {
    #pragma omp sections
    {
      #pragma omp section
      { task_A(); }
      #pragma omp section
      { task_B(); }
      #pragma omp section
      { task_C(); }
      #pragma omp section
      { task_D(); }

    } /* End of work-sharing construct */
  } /* End of parreg: implicit barrier */
  return 1;
}
```

Figure 2.10: OpenMP `section/s` construct

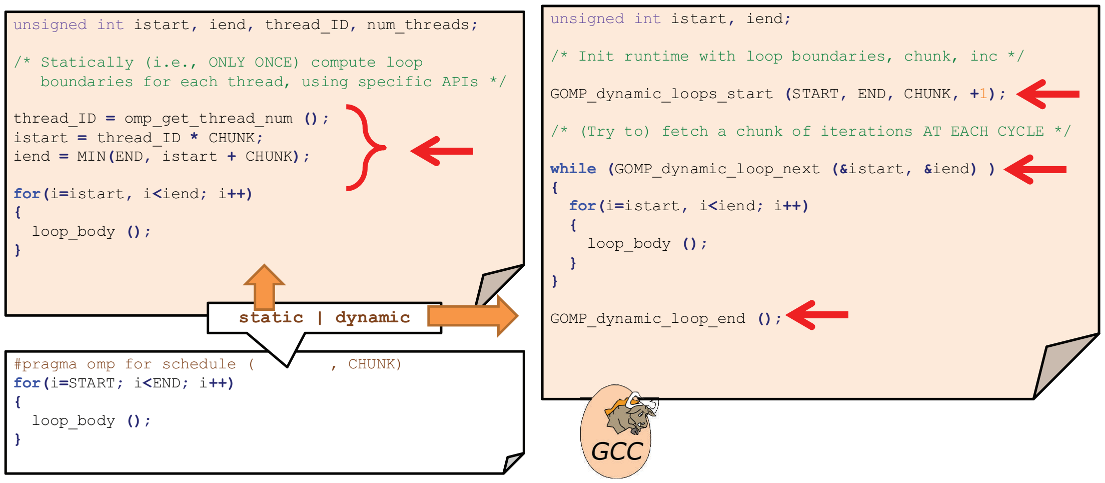


```
int foo()
{
  unsigned int a[16], i;

  #pragma omp parallel num_threads (4) \
    shared (a) private (i)
  {
    #pragma omp for schedule (static, 4)
    for (i=0; i<16; i++)
    {
      a[i] = i;
    } /* End of work-sharing construct */

  } /* End of parreg: implicit barrier */

  return 1;
}
```

Figure 2.11: OpenMP static `for` construct

supports task parallelism in a statically-defined fashion. Each thread request the runtime for one of the tasks isolated by the `pragma omp section` constructs, until all of them have been processed. Figure also shows how the order in which tasks are assigned to parallel thread and *executed* does not necessarily follow the order in which they are encountered (i.e., sections are *written*) in code. Similarly to parallel regions, an implicit barrier for threads is present at the end of each worksharing construct, unless a `nowait` clause is specified.

Worksharing construct are also provided to implement SIMD-like data parallelism. Figure 2.11 shows how a loop can be parallelized so that every worker threads executes a chunk of its iterations. In the example, the `pragma omp for` constructs specifies that each of the thread will be assigned 4 iterations of the loop the works on the array `a`. By specifying the (default) `static` scheduling, iterations are statically assigned to threads *before* entering the loop construct. This can lead to unbalancing in case the amount of work in each iteration is different, harnessing performance. To cope with this, a `dynamic` scheduling scheme is also allowed, under which threads request a new *chunk* of iterations to execute only when the previous one has been completed. Roughly speaking, this scheme implicitly balances the workload at run-time, but such a more complex behavior comes with additional overhead (more runtime calls). This is shown in Figure 2.12, which show the code as transformed by GCC, highlighting (red arrows) the overheads



```
unsigned int istart, iend, thread_ID, num_threads;

/* Statically (i.e., ONLY ONCE) compute loop
   boundaries for each thread, using specific APIs */

thread_ID = omp_get_thread_num ();
istart = thread_ID * CHUNK;
iend = MIN(END, istart + CHUNK);

for(i=istart, i<iend; i++)
{
  loop_body ();
}
```

```
#pragma omp for schedule (         , CHUNK)
for(i=START; i<END; i++)
{
  loop_body ();
}
```

```
unsigned int istart, iend;

/* Init runtime with loop boundaries, chunk, inc */

GOMP_dynamic_loops_start (START, END, CHUNK, +1);

/* (Try to) fetch a chunk of iterations AT EACH CYCLE */

while (GOMP_dynamic_loop_next (&istart, &iend) )
{
  for(i=istart, i<iend; i++)
  {
    loop_body ();
  }
}

GOMP_dynamic_loop_end ();
```

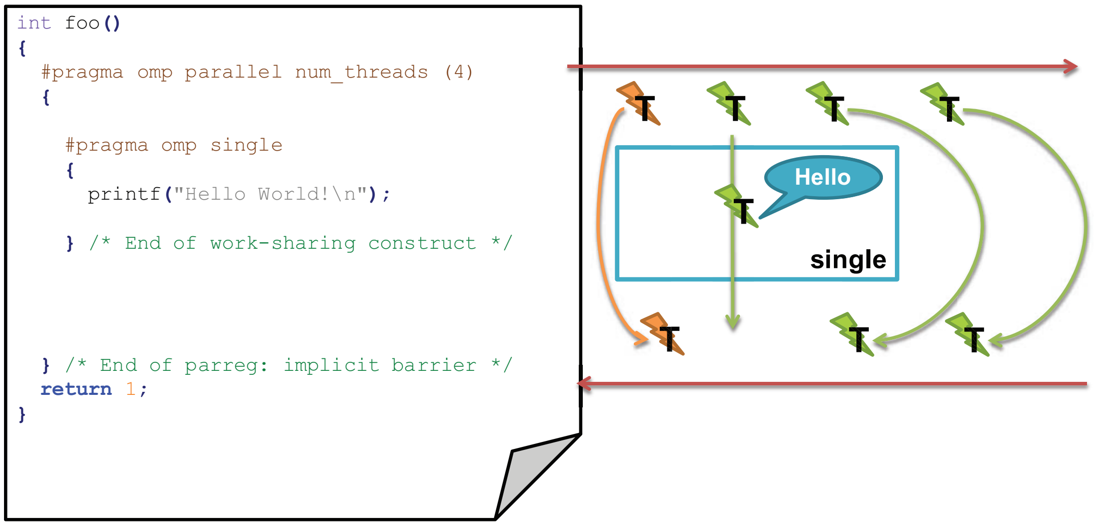
Figure 2.12: Modified GOMP code for loops

introduced by the two *loop* scheduling variants. One last worksharing construct, `pragma omp single`, lets only one thread executing the annotated segment of code. It is shown in Figure 2.13. Similarly, the non-data-sharing `pragma omp master` construct specifies a portion of code to be executed only by the master threads (remember that the thread that creates a parallel region implicitly participates at it – in orange in the examples). The two latter constructs are extremely useful, for instance, in combination with the `task` construct, which will be explained in next section. Worksharing constructs can be closely nested one another, but only with an incurring parallel region.

2.3.2 Tasking

Up to the specification version 2.5 OpenMP used to be somewhat tailored to large array-based parallelism. However, many embedded applications have a lot of potential parallelism which is not regular in nature and/or which may be defined in a data-dependent manner. This kind of parallelism can not be easily expressed with OpenMP 2.5, since its directives lack the ability to dynamically generate units of work that can be executed asynchronously (10).

Nested parallelism can be used in OpenMP 2.5 to express irregular parallelism but this solution has two main drawbacks. First, nesting parallel regions recursively implies the creation of a potentially very high number of threads, which is known to carry a very high overhead. Second, at the end of a `parallel` construct there is an implicit



```
int foo()
{
  #pragma omp parallel num_threads (4)
  {

    #pragma omp single
    {
      printf("Hello World!\n");

    } /* End of work-sharing construct */



  } /* End of parreg: implicit barrier */
  return 1;
}
```

Figure 2.13: OpenMP `single` construct

barrier synchronization which involves all the threads, but which may be unnecessary in the algorithm.

These limitations stem from the fact that OpenMP 2.5 has a thread-centric parallel execution model. Creating additional parallelism can only be done by creating new threads, which has significant cost. Similarly, synchronization can only take place at the thread-team level. The size of a parallel team (i.e., the number of involved threads) is determined upon region creation, and cannot be dynamically modified. This obviously affects the way multiple levels of parallelism can be represented and handled and requires static workload partitioning to prevent poor system usage. Figure 2.14 illustrates this situation. Coarse-grained tasks *S0* and *S1* are initially assigned to two threads, then another two tasks *T* and *U* are encountered, which contain additional parallelism and can be distributed among multiple threads. Picking the same number of threads for both the nested (inner) parallel regions may result in unbalanced execution due to different workload in tasks *T* and *U*. However, there is no means in OpenMP 2.5 to allow threads belonging to a team to migrate to another team, so when threads assigned to task *T* finish their work they just stay idle, waiting for task *U* to complete.

OpenMP 3.0 introduces a task-centric model of execution. The new `task` construct can be used to dynamically generate units of parallel work that can be executed by every thread in a parallel team. When a thread encounters the `task` construct, it prepares a task *descriptor* consisting of the code to be executed, plus a data environment

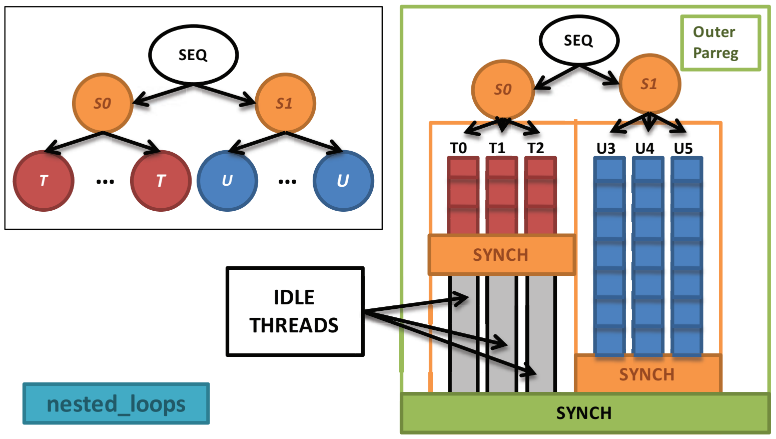
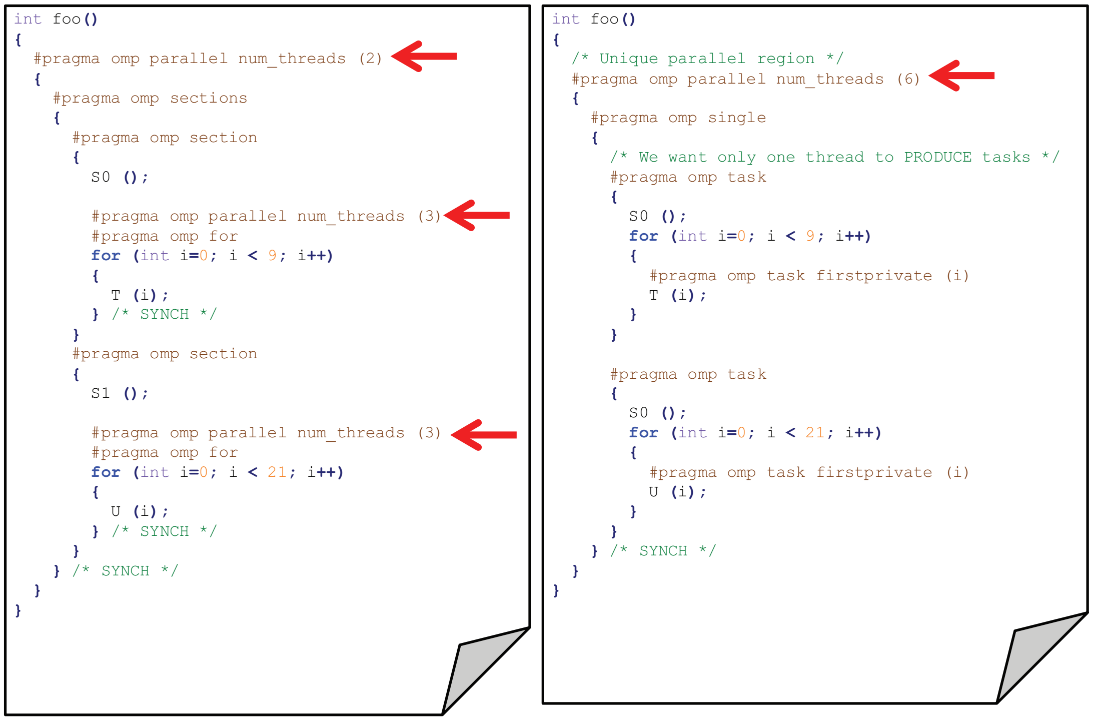


Figure 2.14: Multi-level parallelism using OpenMP 2.5

inherited from the enclosing structured block. `shared` data items point to the variables with the same name in the enclosing region. New storage is created for `private` and `firstprivate` data items, and the latter are initialized with the value of the original variables at the moment of task creation. This means that a snapshot of the status of variables inherited from the enclosing *region* is captured at the instant of task creation. *Shared* variables are not included in the snapshot, thus the programmer is in charge of ensuring their consistency by means of explicit synchronization. The execution of the task can be immediate or deferred until later by inserting the descriptor in a *work queue* from which any thread in the team can extract it. This decision can be taken at runtime depending on resource availability and/or on the scheduling policy implemented (e.g., breadth-first, work-first (42)). However, a programmer can enforce a particular task to be immediately executed by using the `if` clause. When the conditional expression evaluates to *false* the encountering thread suspends the current task region and switches to the new task. On termination it resumes the previous task. Figure 2.15 shows how the same example previously shown (Figure 2.14) can be implemented either using nested parallel regions and a mix of task-based and data-based parallelism (on the left), and with tasking extensions. Besides the aforementioned performance and flexibility issues (the red arrows indicate the – costly – opening of a new parallel region), reader can also notice how the code is much cleaner and simpler in the second case, and moreover, being less “structured” it is easier to maintain and to extend, that is, adding more tasks.



```
int foo()
{
  #pragma omp parallel num_threads (2)
  {
    #pragma omp sections
    {
      #pragma omp section
      {
        S0 ();

        #pragma omp parallel num_threads (3)
        #pragma omp for
        for (int i=0; i < 9; i++)
        {
          T (i);
        } /* SYNCH */
      }
      #pragma omp section
      {
        S1 ();

        #pragma omp parallel num_threads (3)
        #pragma omp for
        for (int i=0; i < 21; i++)
        {
          U (i);
        } /* SYNCH */
      }
    } /* SYNCH */
  }
}
```

```
int foo()
{
  /* Unique parallel region */
  #pragma omp parallel num_threads (6)
  {
    #pragma omp single
    {
      /* We want only one thread to PRODUCE tasks */
      #pragma omp task
      {
        S0 ();
        for (int i=0; i < 9; i++)
        {
          #pragma omp task firstprivate (i)
          T (i);
        }
      }

      #pragma omp task
      {
        S0 ();
        for (int i=0; i < 21; i++)
        {
          #pragma omp task firstprivate (i)
          U (i);
        }
      }
    } /* SYNCH */
  }
}
```

Figure 2.15: Example in Figure 2.14 with *nesting* and *tasking*

The new specifications also enable work-unit based synchronization. The `taskwait` directive forces the current thread to wait for the completion of every children tasks previously generated in the scope of the current task region. This holds only for the immediate successors, not their descendants. This means that a new *task region* also embodies a synchronization context which comprises only the (possible) children of its associated task.

Task scheduling points (TSP) specify places in a program where the encountering thread may suspend execution of the current task and start execution of a new task or resume a previously suspended task. From that moment on, the thread becomes *tied* to that task, and it will be the only responsible for its execution from beginning to end [1]. TSP are found:

1. at `task` constructs;

2. at implicit and explicit *barriers*;

[1]It is possible to avoid this behavior by specifying an `untied` clause, but we we will not consider this possibility for now

3. at the end of the *task region*.

4. at `taskwait` constructs;

Switching tasks on a thread is subject to the *Task Scheduling Constraint*: *"In order to start [...] a new task, this must be a descendant of every suspended task tied to the same thread, unless the encountered TSP corresponds to a barrier region"*(106). This prevents deadlocks when tasks are nested within critical regions. Moreover, the Task Scheduling Constraint guarantees that *"all the explicit tasks bound to a given parallel region complete before the master thread leaves the implicit barrier at the end of the region"*(106). This means that the implicit barrier at the end of a parallel region is enhanced with the semantic of consuming all the previously unexecuted tasks spawned in the parallel region itself before execution can go on. Consequently, all of the threads in the team are potential task consumers, and none of them can be idle while there is still work left to do, as opposite to the implementation in Figure 2.14, where *by construction* some threads cannot contribute to the whole parallel workload. Moreover, since `task` constructs can be directly nested, multiple levels of parallelism can be efficiently exploited, while as explained, traditional work sharing construct cannot be closely nested without the cost of an intervening parallel region.

In Chapter 5 we will analyze in details OpenMP *tasking*, and show how it can be efficiently implemented on shared-memory many-cores.

3

Analysis of the costs of parallel programming support on shared-memory MPSoCs

The ever-increasing complexity of MPSoCs is putting the production of software on the critical path in embedded system development. Several programming models and tools have been proposed in the recent past that aim to facilitate application development for embedded MPSoCs, OpenMP being one of them. To achieve performance, however, it is necessary that the underlying runtime support efficiently exploits the many peculiarities of MPSoC hardware, and that custom features are provided to the programmer to control it. This chapter considers a representative template of a modern multi-cluster embedded MPSoC and present an extensive evaluation of the cost associated with supporting OpenMP on such a machine, investigating several implementation variants that are aware of the memory hierarchy and of the heterogeneous interconnection system.

3.1 Introduction

Advances in multicore technology have significantly increased the performance of embedded Multiprocessor Systems-on-Chip (MPSoCs). Multi-cluster designs have been proposed as an embodiment of the MPSoC paradigm, both in research (54, 68, 144) and industry (15, 46, 70, 101, 109). The availability of such a heterogeneous and hierarchical interconnection system, mixed with the presence of complex on-chip memory hierarchies and – possibly – of hardware accelerators of a different nature obviously has a great impact on application writing (22, 72). Embedded software design for a similar platform involves parallel programming for heterogeneous multiprocessors, under performance and power constraints (140). Being able to satisfy such constraints requires programmers to deal with difficult tasks such as application and data partitioning and mapping onto suitable hardware resources.

OpenMP allows programmers to continue using their familiar programming model, to which it adds only a little overhead for the annotations. Moreover, the OpenMP compiler is relieved from the burden of automatic parallelization and can focus on exploiting the specified parallelism according to the target platform. However, platform-specific optimization cannot be achieved by a compiler only, since OpenMP directives only convey high-level information about program semantics to the compiler. Most of the target hardware specifics are enclosed within the OpenMP runtime environment, which is implemented as a library into which the compiler inserts explicit calls. The radical architectural differences between SMP machines and MPSoCs call for a custom and careful design of the runtime library. The reference GCC-OpenMP implementation (`libgomp`) (52) cannot be of use due to several practical reasons: the small amount of memory available, the lack of OS services with native support for multicore and, above all, a memory subsystem organized as a distributed shared memory, with NUMA latencies within a unique address space.

In particular, different challenges have to be faced locally, at the cluster level and globally, at the system-wide level.

Focusing at the cluster level, an efficient exploitation of the memory hierarchy is key to achieving a scalable implementation of the OpenMP constructs. Particularly, leveraging local and tightly coupled memory blocks to processors (e.g., scratchpads) plays a significant role in:

1. Implementing a lightweight fork/join mechanism;
2. Reducing the cost for data sharing through intelligent placement of compiler-generated support metadata;
3. Reducing the cost for synchronization directives.

In this chapter, several implementation variants are considered for the necessary support to data sharing in OpenMP, that exploit the peculiarities of the memory hierarchy.

Focusing at the system level, that solutions for synchronization and data sharing that are found efficient within a single cluster no longer behave well when considering the platform in its entirety. Different solutions are here studied, which take into account the presence of a hierarchical interconnection system and the strong NUMA effect induced on memory operations by the presence of the NoC.

3.2 Related Work

This chapter presents an extensive set of experiments aimed at assessing the costs and challenges of supporting OpenMP programming on an embedded MPSoC with a large number of cores (up to 64). Authors in (54) propose a 4-cluster system with 17 cores per cluster, implemented on FPGA technology, and a custom programming model (53) for their architecture. Their approach consists in providing the programmer with a small and lightweight set of primitives to support embedded software development. Their methodology requires an expert programmer with a good knowledge of the target platform, while OpenMP ensures a productive and simplified software development process.

Authors of (74, 77) implemented MPI (129) low-level synchronization mechanisms in a topology-aware manner. MagPIe (77) targets wide area systems featuring symmetrical clusters of parallel processing units connected by a fast interconnection. Clusters are connected each other by a high-latency low-bandwidth network, and this architecture perfectly matches the one considered here. Karonis et al. (74) target a multi-level hierarchy of asymmetric clusters connected by a Wide Area Network (WAN). The approach presented here here can be seen as a specialization of theirs for a single level hierarchy of clusters. All these implementations are built upon the send/receive MPI primitives provided by an existing runtime –MPICH (57)–, while, on the contrary, the presented runtime runs directly on bare metal. In fact the proposed APIs do not implement a specific programming model/specification and do not rely on another layer, rather provides a fast and generic low-level API set for threads synchronization on a clustered environment. Moreover, even though the clusters targeted here are made symmetric processing units, the synchronization mechanisms could be extended also to asymmetric ones with no significant modifications.

Several researchers have investigated the adoption of OpenMP for embedded MPSoCs with a similar memory model. However, previous work in this field either is very specific to a particular platform, or lacks a detailed analysis of performance implications of OpenMP programming patterns on the underlying hardware. Authors of (67) present an OpenMP implementation for a Cradle CT3400 OS-less MPSoC. They provide optimized implementation of the `barrier` directive, which is used as a direct term of comparison in Section 3.5. An extended OpenMP programming framework for the Cradle 3SoC platform is described by Liu et al. in (90)(91). Custom directives are provided to enable parallel execution on DSPs and to exploit specific banks of the memory hierarchy for data placement. The necessity of extensions to OpenMP to make it a viable programming model for embedded MPSoCs are also discussed in (33, 69). In (33) Chapman et al. suggest that directives to specify execution on accelerators should be necessary, similar to those proposed by Liu, as well as language feature to specify the priority (69) of a given thread to discriminate between more or less critical activities. Authors agree that data attribute extensions may help the compiler make good decisions on where to store data, but no practical solution is discussed, nor implementations are proposed. An initial implementation of a standard OpenMP programming interface is however provided for a TI C64x+ -based MPSoC with a multi-level memory hierarchy. This architecture closely resembles the generic cluster template targeted in this chapter. However, the authors do not provide a detailed evaluation of the implementation.

Extensions to OpenMP to enable data distribution have been proposed in the past in the High Performance Computing domain (18, 33, 103). These proposals are closely related in the choice of exposing features for locality-aware data placement at the programming model level. On the other hand, the differences at the architectural level between traditional NUMA multi-processors and embedded MPSoCs are very significant and imply a completely different set of challenges and viable solutions. In particular, in traditional NUMA machines (computer clusters) inter-node communication takes orders of magnitude longer than local operations, large enough to hide the cost of virtual memory paging. This approach cannot be considered for two reasons. First, the target architecture lack the necessary hardware and software support (i.e., per-core

MMUs and full-fledged operating systems). Second, all communication travels on-chip, where latency is much lower and bandwidth is much higher, which would no longer compensate for the high cost of memory paging.

3.3 Target Architecture

The simplified block diagram of the target cluster architectural template is shown in Figure 3.1. The platform consists of a configurable (up to 16) number of processing elements (PEs), based on a simplified (RISC-32) design without hardware memory management. The interconnection network is a cross-bar, based on the STBus protocol. Support for synchronization is provided through a special hardware semaphore device.

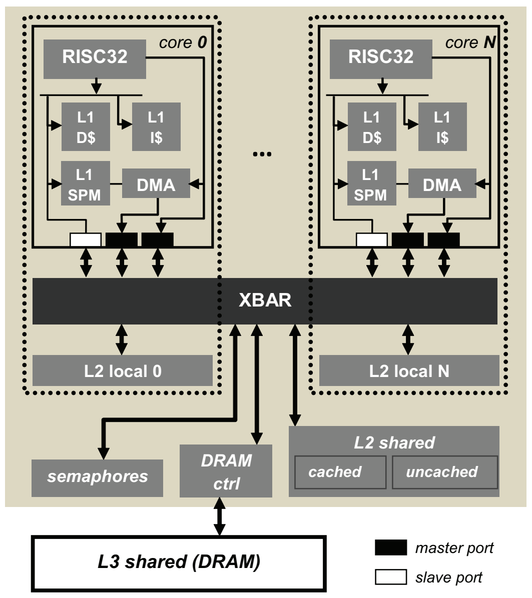


Figure 3.1: Target cluster template

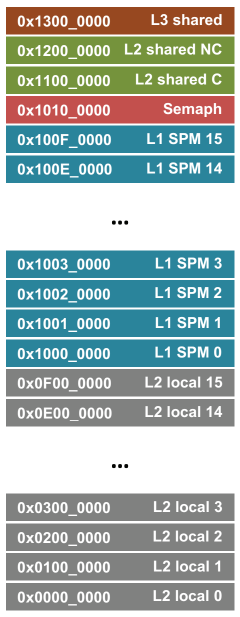


Figure 3.2: PGAS

The memory subsystem leverages a Partitioned Global Address Space (PGAS) organization. All of the on-chip memory modules are mapped in the address space of the processors, globally visible within a single shared memory space, as shown in Figure 3.2. The shared memory is physically partitioned in several memory segments, each of which is associated (i.e., tightly coupled, or placed in close spatial proximity) to a specific PE. Each PE has an on-tile L1 memory, which features separate instruction and data caches, plus scratchpad memory (SPM). Remote L1 SPMs can be either directly accessed or through on-tile Direct Memory Access (DMA) engines. Each PE is logically associated to a local L2 memory bank, where by default program code and data private to the core are allocated. Local L2 memory is only cacheable in a local L1 cache. Accessing the local L2 memory of a different PE is possible, but requires appropriate cache control actions. Processors can also directly communicate through the L2 shared memory, which features both cacheable and non-cacheable banks. Data allocated in the cacheable bank can be cached by every processor, therefore multiple copies of the same shared memory location may exist simultaneously in the L1 caches. Consistent with the OpenMP *relaxed-consistency* memory model, cache coherence is managed through software flush instructions in the runtime library. The off-chip shared L3 DRAM memory is also mapped in the address space of processors.

The proposed template captures several design choices proposed in recent embedded MPSoCs, such as the Texas Instruments TNETV3020 (125) and TMS320TCI6488 (124).

Figure 3.3 shows the reference Multi-Cluster MPSoC, featuring 4 clusters interconnected through a Network on Chip based on (16). The NoC topology is a mesh (2$\times$2); it is a wormhole network: packets are divided into a sequence of flits, which are transmitted over physical links one by one in pipeline fashion. Every switch has several ports. Every port has two input and six output buffers. A flit is transmitted only when the receiving port has free space in its input buffers. Every request of transaction is split into several flits on the NoC. The path followed by the flits is decided by the Network Interface (NI) (see Figure 3.1).

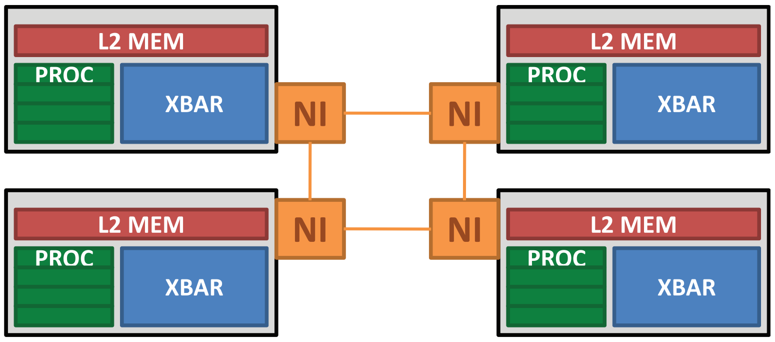


Figure 3.3: 2x2 Cluster Architecture

The network interface catches every non-local access and emits it on the NoC, which forwards it to the NI of the destination cluster. Then the local Master port accesses the local target address.

Finally, the overall address space of the multi-cluster platform is also organized as a PGAS, but the effect of NUMA latencies to access memories from a different cluster are even more pronounced due to the necessity of traversing the NoC.

3.4 OpenMP Support Implementation

An OpenMP implementation consists of a code translator and a runtime support library. The framework presented here is based on the GCC 4.3.2 compiler, and its OpenMP translator (GOMP (52)). Most of the platform-specific optimizations are enclosed in the runtime library, which – on the contrary – does not leverage the original GCC implementation. In the remainder of this section explains the needed modifications to the compiler and runtime to achieve functionality and performance on the generic MPSoC architectural template presented in Section 3.3.

3.4.1 Execution Model

OpenMP adopts the fork-join model of parallel execution. To support this, the GCC implementation of the runtime library (`libgomp` (52)) leverages the Pthreads library to dynamically create multiple instances of the outlined functions. Pthreads require abstraction layers that allow tasks on different cores to communicate. Inter-core communication on embedded MPSoCs requires specific support, and has significant associated overheads (33).

For this reason, the `libgomp` library cannot be ported *as-is* on the target architecture, and the runtime environment was re-designed from scratch, implementing it as a custom lightweight library where the master core is responsible for orchestrating parallel execution among available processors. Rather than relying on dynamic thread creation master and slave threads are statically allocated to the processors. At boot time the executable image of the program+library is loaded onto each processor local L2 memory. When the execution starts all processors run the library code. After a common initialization step, master and slave cores execute different code. Slave cores immediately start executing a *spinning* task, where they busy wait for useful work to do. The master core starts execution of the OpenMP application.

When a parallel region is encountered, the master points slave cores to the outlined parallel code and to shared data. At the end, a global barrier synchronization step is performed. Slave cores re-enter the *spinning* task, while the master core jumps back to the execution of the main application, thus implementing the join mechanism.

The *spinning* task executed by the slaves while not in parallel regions must be implemented in such a way that it does not interfere with the execution of sequential parts of the program on the master core. Polling or signaling activity should not inject significant interfering traffic on the interconnect. To ensure this, a message exchange mechanism is adopted, where the slave cores spin on a local queue. Queues are implemented as buffers residing on the local L1 SPM of every slave core, so transactions generated by polling activity never enter the system interconnect. Upon entrance into a parallel region the master sends a message containing task and frame pointers in the queues of all slave cores.

It must be pointed out that most of the OpenMP implementations leverage on the same set of threads to execute all parallel regions during the entire application (featuring the so-called *Thread Parking*). Dynamically spawning threads at run-time in the target architecture has a significant cost for synchronization and communication. Moreover, here mainly image processing domain are targeted, which leverages SIMD and SPMD applications. Their control flows consist of parallel loops with no branches, thus naturally fitting a parallel runtime based on parking instead than dynamic thread spawning.

Key to minimizing the overhead associated with the join mechanism is the choice of a lightweight barrier algorithm, which is discussed in the following section.

3.4.2 Synchronization

OpenMP provides several mechanisms to synchronize the parallel threads, with `atomic`, `critical` and `barrier` being the most important. While `critical` and `atomic` sections can be straightforwardly implemented on top of hardware *test-and-set* semaphores, the `barrier` directive deserves more attention. In the OpenMP programming model barriers are often implied at the end of parallel regions or work-sharing directives. For this reason they are likely to overwhelm the benefits of parallelization if they are not carefully designed to account for hardware peculiarities and potential bottlenecks.

This section focuses on intra-cluster synchronization, while global synchronization across the entire platform is discussed in Section 3.4.2.1.

Several implementations of OpenMP for MPSoCs adopt a centralized shared barrier (67, 90, 91). This kind of barrier relies on shared entry and exit counters, which are atomically updated through lock-protected write operations. In a centralized barrier algorithm, each processor updates a counter to indicate that it has arrived at the barrier and then repeatedly polls a flag that is set when all threads have reached the barrier. Once all threads have arrived, each of them is allowed to continue past the barrier.

A serious bottleneck arises with this algorithm, since busy waiting to test the value of the flag occurs on a unique shared location. The *Master-Slave* barrier algorithm works around this problem by designating a *master* processor, responsible for collecting notifications from other cores (the *slaves*). Since this communication happens through distinct memory locations the source of contention of the shared counters is removed. The master-slave barrier is structured in two steps. In the *Gather* phase, the master waits for each slave to indicate its arrival on the barrier on a private status flag. After arrival notification slaves poll on a separate private location. In the *Release* phase of the barrier, the master broadcasts a termination signal on each slave's polling flag.

The implementation of the Master-Slave barrier must reflect two aspects:

1. During the *gather* phase the master core polls on memory locations through which slaves indicate their arrival. During the release phase the slave cores poll on memory locations through which the master notifies them of barrier termination. Even if all these memory locations are distinct, significant contention can still arise if all are hosted on the same memory device (we refer to this implementation as a *master-slave shared* barrier in the following). Indeed the source of contention has only been shifted from the memory cell to the memory port.

2. A situation in which traffic generated by polling activity of cores is injected through the system interconnect towards shared memory locations potentially leads to congestion. This may happen when the application shows load imbalance in a parallel region, or when constructs such as `master` and `single` induce

a single processor to perform a useful job while the others (typically) wait on a barrier.

In (33), Chapman et al. leverage a master-slave barrier algorithm for their OpenMP implementation, but encounter similar problems to those described above. To address these issues, the *distributed* implementation of the master-slave barrier considered here leverages L1 scratchpads. Specifically, master and slave poll flags are allocated onto the respective local L1 SPM and they are accessed through message exchange. In this way the number of messages actually injected in the interconnect is limited to $2 \times (N - 1)$, where N is the number of cores participating in a barrier operation.

The direct comparison of these barriers is shown in Figure 3.4.

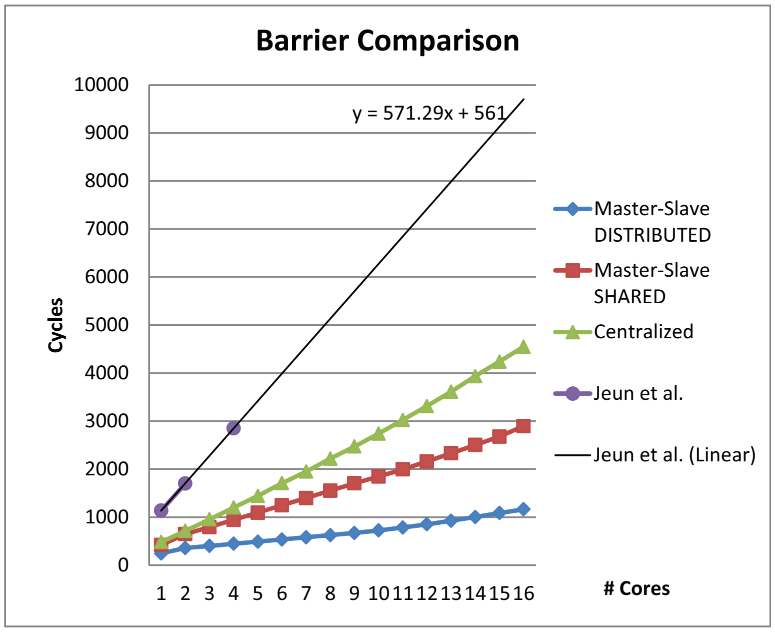


Figure 3.4: Cost of barrier algorithms with increasing number of cores.

The centralized shared barrier provides the worst results. The cost to perform synchronization across 16 cores with this algorithm is around 4500 cycles. The behavior of the barrier is linearly dependent on the number of cores N, so it has a dependency of $\approx 270 \times N$ from linear regression. The high cost of this barrier algorithm is not surprising, and is in fact cheaper than similar implementations. As a direct term of comparison reported here results published by Jeun and Ha (67) for two variants of the centralized barrier implementation on an embedded MPSoC, which show trends of $\approx 725 \times N$ (original) and $\approx 571 \times N$ (optimized).

The master-slave shared barrier mitigates the effects of the bottleneck due to the shared counter. The cost for synchronizing 16 cores is reduced to ≈ 3000 cycles (*gather* + release), and linear regression indicates a slope of $\approx 150 \times N$.

Employing a distributed algorithm completely removes the traffic due to busy-waiting, which significantly reduces the cost of the barrier. Synchronization among 16 cores requires around 1100 cycles, with a tendency of $\approx 56 \times N$.

Plots shown in Figure 3.4 were obtained by only executing barrier code in the system. This is clearly a best case for the barrier performance, since there is no other interfering traffic competing for system resources. To investigate the impact of different barrier algorithms on real program execution results are given, for three benchmarks that emphasize the three representative use cases discussed above.

1. **`#pragma omp single`:** When the `single` directive is employed only one thread is active, while the others wait on the barrier. This behavior is modeled with a synthetic benchmark in which every iteration of a parallel loop is only executed by the first encountering thread.

2. **Matrix multiplication:** A naive parallelization of the fox algorithm for matrix multiplication, which operates in two steps. Each processor performs local computation on submatrices in parallel, then a left-shift operation takes place, which cannot be parallelized and is performed by the master thread only. The `master` block must be synchronized with two barriers, one upon entrance and one upon exit.

3. **Mandelbrot set computation:** This benchmark is representative of a common case in which parallel execution is not balanced. The main computational kernel is structured as a doubly nested loop. The outer loop scans the set of complex points, the inner loop determines – in a bounded number of iterations – whether the point belongs to the Mandelbrot set. Since the iteration counts are not equal (and possibly very different) for every point, parallelizing the outermost loop with static scheduling leads to unbalanced threads.

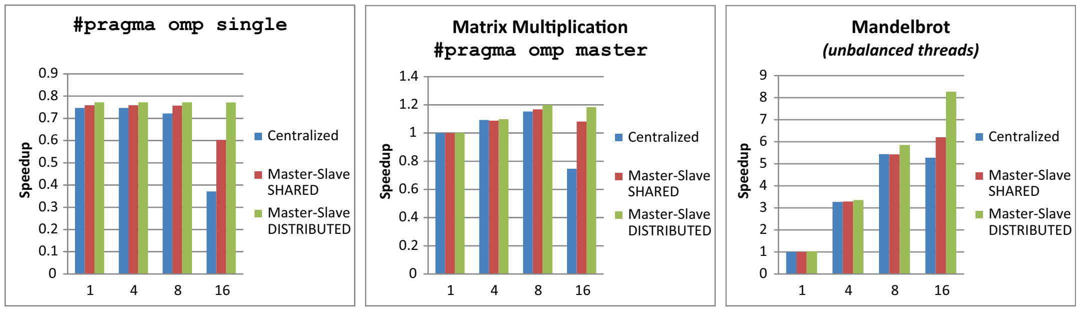


Figure 3.5: Impact of different barrier algorithms on real programs.

Results for each of these benchmarks are shown in Figure 3.5. The plots confirm that the barrier implementation has a significant impact on real programs adopting common programming patterns such as `single` and `master` sections. Focusing on the synthetic benchmark, it is possible to notice that the distributed master-slave barrier allows the `single` directive to scale perfectly with an increasing number of processors. On the contrary, the shared master-slave barrier and – in particular – the centralized barrier degrade significantly program performance when the number of cores increases. An analogous behavior can be seen in the *Matrix Multiplication* benchmark, where a significant portion of the parallel loop is spent within the `master` block. The same effect can be observed in Mandelbrot, and, more in general, whenever it is impossible to ensure perfect workload balancing from within the application.

3.4.2.1 Barrier synchronization in the clustered architecture

Figure 3.6 shows the linear regression of the cost of the three barriers so far for up to 64 processors. This is clearly an optimistic projection, since it is based on the results achieved with the single cluster, and does not account for the effect of the NoC for inter-cluster communication, and does not considers effects of increased contention. Focusing on 64 cores, notwithstanding the optimistic content of the plot, even the distributed Master-Slave barrier, the only one worth considering so many cores, has a significant cost. Indeed such a costly support for synchronization prevents efficient execution of fine-grained parallelism. Things get even worse when actually execute the distributed barrier algorithm on the multi-cluster architecture. Figure 3.7 shows the actual cost of the Master-Slave barrier as a function of the number or processors in the system. As explained in Section 3.3 the targeted architectural template consists of four clusters,

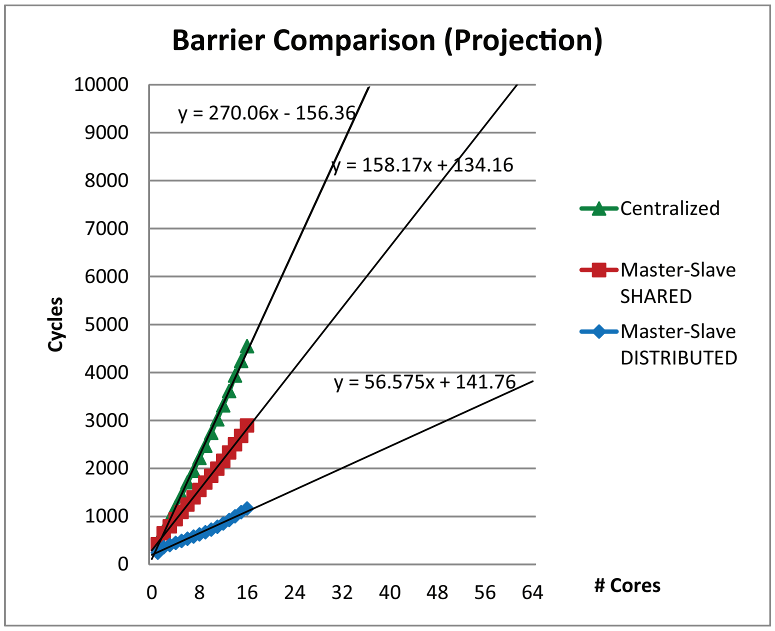


Figure 3.6: Projections for barriers cost in a 64 processor system

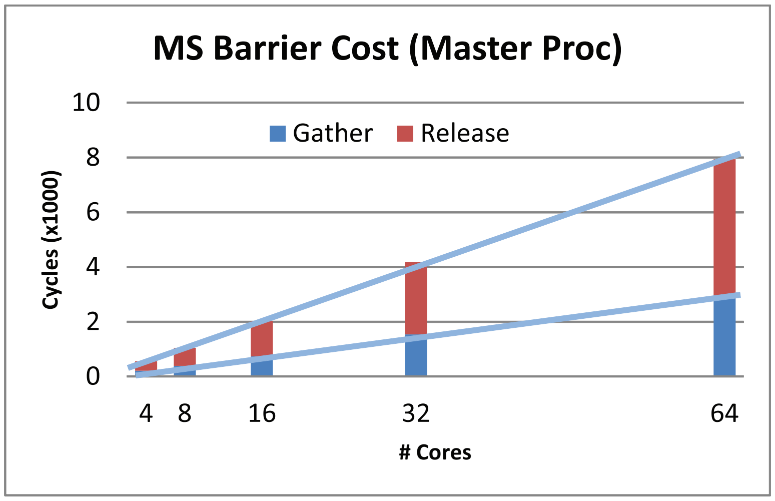


Figure 3.7: Measured cost of the distributed Master-Slave barrier on a cluster architecture

each featuring an equal (parameterizable) number of cores. Considering 1, 2, 4, 8, 16 cores per cluster the system configurations has 4, 8, 16, 32, 64 cores considered in the X axis of Figure 3.7. The actual cost to synchronize 64 cores with the distributed master-slave barrier is twice as large as expected from the projection. The increase in

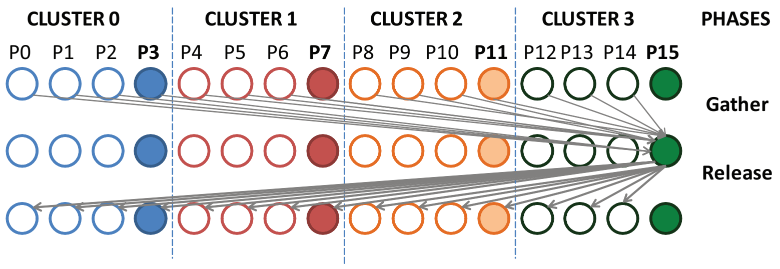


Figure 3.8: MS Barrier in a Clustered Architecture

the cost of this barrier is due to two factors:

1. All cores have to communicate directly with the unique master in the system (and vice-versa, see Figure 3.8), and they are subject to both non-uniform latencies depending on their physical distance from the master and to priority arbitration at the NI.

2. The master core is in charge of managing an increasing number of slaves, which results in increased processing time (besides memory effects)

Both effects could be alleviated by adopting a barrier algorithm that introduces an additional step in which cores synchronize locally prior to communicating with the master core. The tree barrier algorithm (139) (97) (136) is a 2-phase multi-stage

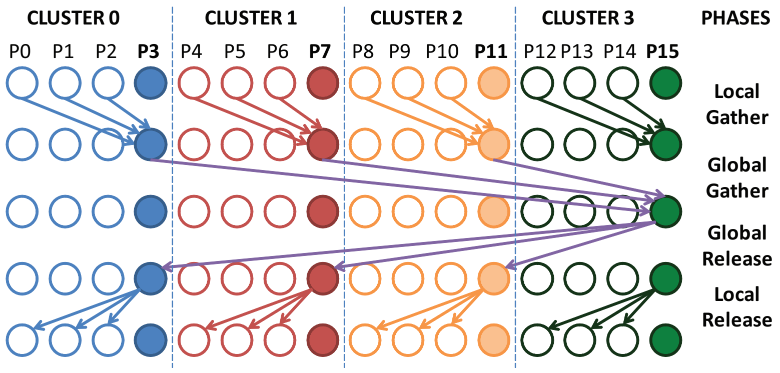


Figure 3.9: 2-stage Tree Barrier in a Clustered Architecture

synchronization mechanism. It can be seen as an extension of a standard Master-Slave barrier, since it features a *gather* phase followed by a Release phase. Each phase is however divided in a sequence on stages. In each *gather* stage a few processors (Cluster Masters) are in charge of collecting each one a subset of the others (Slaves). Then at the following stage the Cluster Masters become Slaves to the unique Global Master for a global *gather* phase. The Release phase is specular. In Figure 3.9 a 16 processors Tree barrier is depicted, with 4 Local Masters and 2 stages/phase. As shown, the Processors set is partitioned so that for each subset a Local Master Processor is chosen: P3 controls P0, P1 and P2; P7 controls P4, P5 and P6, and so on. Each Local Master collects its Local Slaves (each subset has a different color). In the second, and last, *gather* stage, P3, P7 and P11 become Slaves and P15, the so-called the Global Master, collects them. The Release phase follows, where we see P15 Releasing P3, P7 and P11 which will themselves release their Local Slaves.

The Tree Barrier algorithm is cluster-aware and essentially acts as a Tournament Barrier (61). The difference is that this implementation features two Stages (inter- and intra-cluster), while in the standard implementation the tournament features Log2N Stages. This means that for instance at the intra-cluster Stage all processors within each cluster fight against each other, the winner being the Local Master. The overall algorithm features four phases (Local and Global, *gather* and Release) for any number of processors/clusters, thus performance scales perfectly with them. The tree barrier limits the number of exchanged messages over the NoC to $2 \times (C - 1)$, where C is the number of clusters in the system. Furthermore, the Global Master has the sole responsibility of managing $C-1$ Global Slaves, while local *gather* takes place in parallel over different clusters. These benefits compensate for the additional stages needed. Following listing shows barrier code executed by the local master.

```
void local_master_code(int myid)
{
  int i;
  /* Gather Local Slaves */
  for (i=local_slaves_istart; i<local_slaves_iend; i++)
    while (!*((bool*) tree_bar_master_flags_local[i]));

  /* Notify to Global Master */
  *((bool*) tree_bar_master_flags_global[myid]) = TRUE;

  /* Remote Wait */
  while (!*((bool*) tree_bar_slaves_flags_global[myid]));

  /* Release Local Slaves */
  for (i=local_slaves_istart; i<local_slaves_iend; i++)
    *((bool*) tree_bar_slave_flags_local[i]) = TRUE;
}
```

Listing 3.1: Local Master Code for the Tree Barrier

Figure 3.10 shows the cost for the tree barrier running on a 4-cluster platform with 4, 8, 16, 32 and 64 processors (respectively 1, 2, 4, 8 and 16 processors per cluster, the highest ID-ed being the Local master). For the 4-processor configuration the *local gather* stage is empty (no Local Slaves to wait for).

Figure 3.10 shows both results for Local and Global Masters and provide the breakdown of the cost of each stage. Thus, for Local Master the *local gather*, *global notify*, *global wait* and *local release* timings are plotted, while for the Global Master we see *local gather*, *global gather*, *global release* and *local release*. This plot demonstrates that

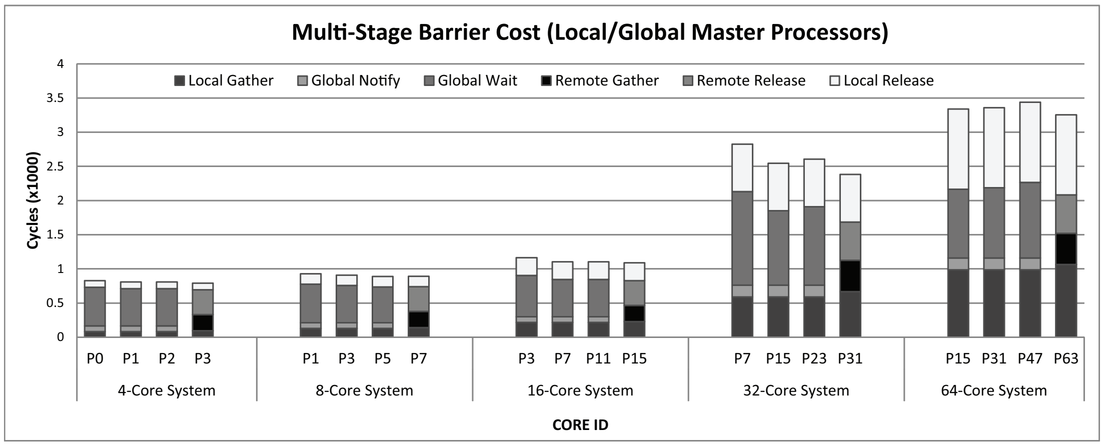


Figure 3.10: Tree Barrier Cost, from left to right, for 4, 8, 16, 32, 64 processors

splitting synchronization into *local gather* operations and reducing the number of messages traveling through the NoC has a dramatic effect on the barrier cost. With this implementation 64 cores can be synchronized within roughly 3500 cycles. While it still does not enable very fine-grained parallelization if frequent barrier synchronization is required, this result is the best achievable with a software implementation. This results suggest however that hardware support to global synchronization in many-cores will be necessary to enable fine-grained data parallelism (136).

3.4.3 Data Sharing and Memory Allocation

OpenMP provides several clauses to specify the sharing attributes of data items in a program, which can be broadly classified into *shared* and *private* types. The classification depends on whether each parallel thread is allowed to reference a private instance of the datum (`private`) or they must be ensured to reference a common memory location, be it through the entire parallel region (`shared`) or only once at its beginning/end (`firstprivate`/`lastprivate`, `reduction`).

When a variable is declared as `private` within a parallel region the GOMP compiler duplicates its declaration at the beginning of the parallel region code. Each thread thus refers to a private copy of the variable. This behavior can be implemented *as is* in the target platform, since private data is allocated by default onto local L2 memories to each core, thus ensuring the correct semantics for the `private` clause.

Shared data items are typically declared out of parallel regions, within the scope of the function enclosing the `parallel` construct. This part of the program is only executed by the master core, thus implying that shared variables are by default allocated on the stack of the master thread.

```
int foo()
{
  /* Shared variable lives in master thread's stack */
  double A[100];
  int i;

#pragma omp parallel for shared(A) private(i)
  for (i=0; i<N; i++)
    A[i] = f(i);
}
```

A common solution to make shared variables visible to slave threads[1] is to rely on a sort

[1]which only exist within a parallel region

of marshalling operation in which the compiler generates metadata containing pointers to shared data. Specifically, the compiler collects shared variable declarations into a C-like `typedef struct`.

```
/* Compiler-generated metadata */
typedef struct
{
  double[100] *A;
} omp_data_s;
```

Before entering a parallel region the master core stores the addresses of shared variables into metadata, then passes the structure's address to the runtime environment, which in turn makes it available to the slaves.

```
int foo()
{
  double A[100];
  omp_data_s mdata;

  /* Metadata points to shared data */
  mdata.A = &A[0];

  /* Then its address is passed to the runtime */
  GOMP_parallel_start (foo.omp_fn0, &mdata);
}
```

Finally, the compiler replaces all accesses to shared variables within the outlined parallel function with references to the corresponding fields of the metadata structure.

```
int foo.omp_fn0 (omp_data_s *mdata)
{
  int i;

  for (i=LB; i<UB; i++)
     /* Replace shared var accesses with metadata alias */
    (mdata->A)[i] = f(i);
}
```

On a MPSoC such as the considered one, efficiently implementing data sharing is tricky due to the complex memory hierarchy. As explained in Section 3.3 each core features a local bank of memory (L2 local) onto which stack/private data is by default allocated. Local L2 memory to a core can be accessed by other processors, but the access latency is non-uniform, since it depends on the physical distance of the core from the memory bank, the degree of contention for the shared resource and the level of congestion in the interconnect. This default data sharing implementation solution is the baseline for investigations, and it will be later referred to as **Mode 1**. Here slave processors access both shared data and metadata from the master core local L2 memory. Since this memory bank also hosts all master core private code and data, it is delayed by other processor activity on memory, as shown in Figure 3.11(a). To overcome this bottleneck, a set of compiler-directed placement alternatives are explored, that take into account the memory subsystem organization.

The first variant consists in exploiting the L1 SPM local to each core to host private replicas of metadata. Since metadata contains read-only variables no inconsistency issues arise when allowing multiple copies. The custom GCC compiler modifies the outlined parallel function code in such a way that upon entrance into a parallel region each core initiates a DMA copy of metadata towards its L1 SPM.

```
int foo.omp_fn0 (omp_data_s *mdata)
{
  int i;
  int *local_buf;

  /* Allocate space in local SPM to host metadata */
  local_buf = SPM_malloc (sizeof (omp_data_s));

  /* Call runtime to initiate DMA */
  __builtin_GOMP_copy_metadata (mdata, local_buf);

  /* Point to local copy of metadata */
  mdata = local_buf;

  for (i=LB; i<UB; i++)
    (mdata->A)[i] = f(i);
}
```

This solution removes all traffic towards the master core L2 local memory due to accesses to metadata (see Figure 3.11(b)), and will be later referred to as **Mode 2**.

Since most memory traffic during parallel regions is typically due to shared variable accesses, in the second placement variant the compiler checks for variables annotated with sharing clauses and re-directs their allocation to the non-cacheable segment of the shared L2 memory. This placement scheme is called **Mode 3** (see Figure 3.11(c)).

Mode 4 combines **Mode 2** and **Mode 3**: Metadata is accessed from L1 and shared data from shared L2 memory.

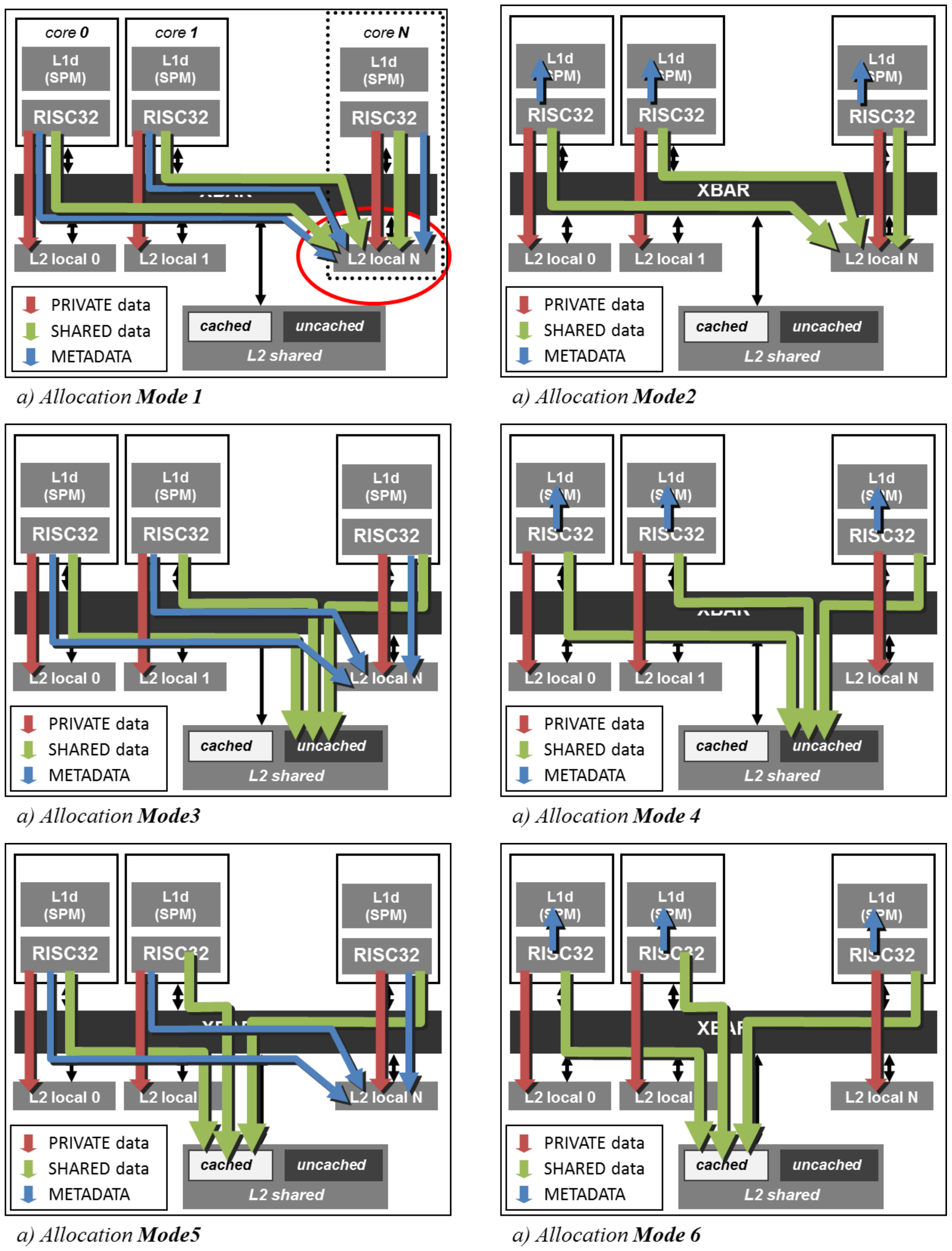


Figure 3.11: Data and metadata allocation strategies (modes)

When the number of processors increases and the program exhibits significant activity on shared data another bottleneck arises. Multiple concurrent requests are serialized on the port of the shared L2 memory. The use of the cache may clearly alleviate this problem. Indeed, many OpenMP applications exploit data parallelism at the loop level, where shared arrays are accessed by threads in (almost) non-overlapping slices. Besides improving data locality, the use of (coherent) caches allow to allocate separate array portions on different memories, thus eliminating the source of the bottleneck. To investigate this effect shared data can be placed on a cacheable region of the shared L2 memory. If metadata resides on the master core local L2 the placement scheme is called **Mode 5**. If metadata is replicated onto every L1 SPM it's called **Mode 6**.

3.4.3.1 Extensions for Data Sharing and Distribution on a cluster-based architecture

The memory model of the clustered MPSoC still adheres to the Partitioned Global Address Space paradigm. Indeed each of the memory banks hosted on every cluster is mapped onto an unique system-wide address, so that each processor can directly access every memory location. Clearly, the presence of a heterogeneous communication medium makes the effect of non-uniform memory access cost (in terms of increased latency and decreased bandwidth) even more important. It is therefore quite obvious that the problems about data sharing discussed in the previous section will be amplified when considering a clustered architecture. Indeed contention for shared data from a unique memory device in the system will be subject to several sources of architectural non-homogeneity, which eventually hinder execution of OpenMP parallelism:

1. massively increased contention;
2. NUMA latencies;
3. effect of the arbitration policy at the NI.

Array partitioning techniques (32)(18)(34) have been proposed in the past in the high performance computing domain to address a conceptually identical problem: efficiently programming NUMA machines (computer clusters). Data distribution is the key technique to ensure locality of computation and affinity between threads and memory. In (94), Marongiu et al. investigated the effect of implementing a data distribution techniques on a scratchpad-based MPSoC very similar to the basic cluster template targeted here. The principal use of distribution in that case was efficient exploitation of SPM space, and to improve data locality. The beneficial effect of data distribution to relieve the pressure of high contention on a single memory device was not addressed in that previous work, due to the small number of processors available in the system (up to 8). They provide the typical **block** and **cyclic** distribution schemes found in similar approaches through a custom extension to the OpenMP API and compiler. While profile-based techniques such as those described in (94) can clearly be applied to improve the efficiency of distribution, they are not the subject of this investigation.

The programmer can trigger array partitioning in the compiler through the custom `distributed` directive.

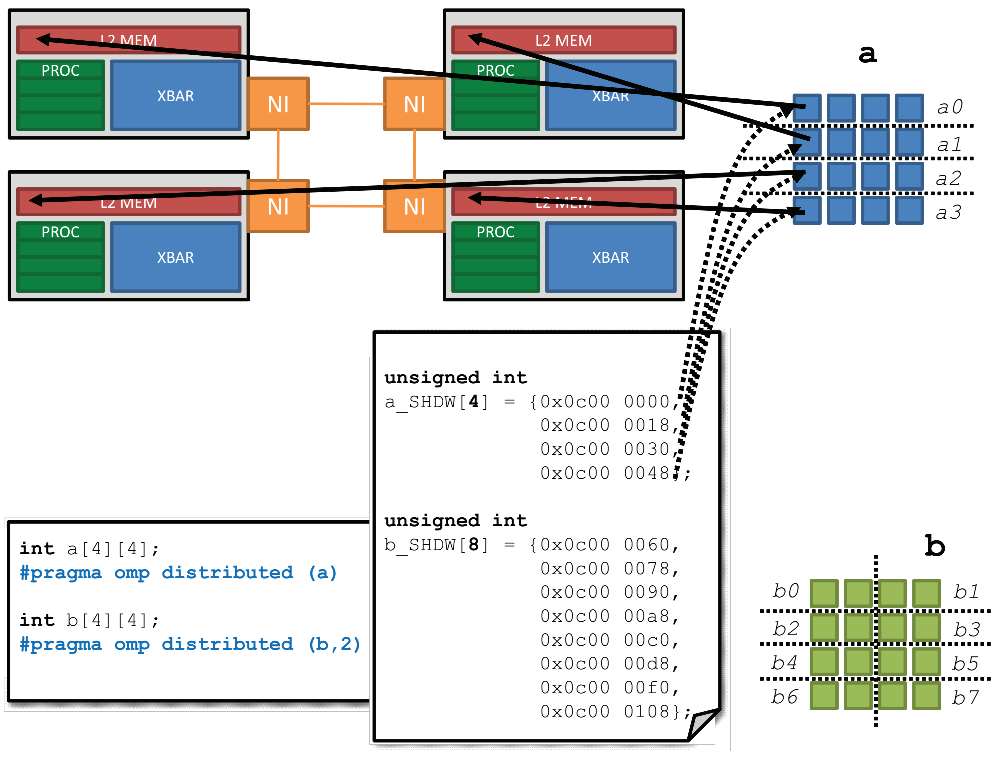
```
double A[100];
#pragma omp distributed (A[, tilesize])
```

The optional *`tilesize`* parameter is used to specify the granularity of partitioning, namely the size – expressed in terms of contiguous array elements – of the elementary tile. If this parameter is not specified the compiler generates as many tiles as available cores, thus implementing **block** distribution. If the parameter is given, tiles are sorted out to memories in a round-robin fashion, thus implementing **cyclic** distribution.

To customize the architectural setup a few additional flags were implemented in the compiler, to specify the number of clusters and cores in the system, the base address for the distributed shared memory space, and the offset between consecutive memory nodes. These flags support flexible compilation of code for different architectural templates.

The implementation of data distribution does not rely on traditional techniques based on OS support for virtual paging and page migration, as such a heavyweight software abstraction would be too costly for an embedded system such, and would easily overwhelm the benefits of improved data locality and reduced contention. The technique proposed here, rather improves the software address translation described in (94).

The parser, the gimplifier and the OpenMP lowering and expansion passes in GCC were modified to create a *shadow* copy of each `distributed` array in the program. This shadow array contains the addresses of the tiles corresponding to the requested



```
int a[4][4];
#pragma omp distributed (a)

int b[4][4];
#pragma omp distributed (b,2)
```

```
unsigned int
a_SHDW[4] = {0x0c00 0000,
             0x0c00 0018,
             0x0c00 0030,
             0x0c00 0048};

unsigned int
b_SHDW[8] = {0x0c00 0060,
             0x0c00 0078,
             0x0c00 0090,
             0x0c00 00a8,
             0x0c00 00c0,
             0x0c00 00d8,
             0x0c00 00f0,
             0x0c00 0108};
```

Figure 3.12: Implementation of data distribution in a multi-cluster MPSoC

partitioning scheme (granularity and type). Figure 3.12 shows a pictorial representation of this process.

Each reference to a distributed array in the program is transformed into three basic operations:

1. computation of the target tile corresponding to the current reference offset

2. a lookup in the *shadow* array to retrieve the base address of the target tile

3. sum of the proper offset to address the correct element within the tile

In (94), the compilation process was split in two parts. Distributed array references in the OpenMP application were instrumented by referencing an `extern` data structure, conceptually equivalent to the *shadow* arrays described here. This data structure was actually generated by a separate compilation process based on profile information, and finally linked with the OpenMP program object code. The main drawback of this approach consists in the fact that `extern` objects obviously escape the optimization process in the compiler. For this reason, instrumented accesses to distributed arrays in (94) could not be optimized. Here, the declaration of shadow arrays is inlined in the OpenMP program code, which allows GCC optimizers to polish `distributed` array reference expressions.

3.5 Experimental Results

This section presents the experimental setup and the results achieved. An instance of the MPSoC template described in Section 3.3 has been implemented within a SystemC full system simulator (92). The architectural parameters are detailed in Table 3.1. All implementation variants described in the previous sections are validated on several

Processor	*RISC* @200MHz
Interconnect	Hierarchical: Crossbar (*STBus*) + NoC ($\times$*pipes*)
L1 I-cache	4KB, direct mapped, latency: 1 cycle
L1 D-cache	4KB, 4way set-assoc, latency: 1 cycle
L1 SPM	16KB, latency: 1 cycle (local), 10 cycles (remote)
L2 local/shared	latency: 5 cycles (local), 15 cycles (remote)
remote L2	variable (single-hop traversal: 10 cycles)

Table 3.1: Architectural parameters

benchmarks from the *OpenMP Source Code Repository* (41) benchmark suite. For simplicity and clarity of discussion the exposition is divided in two subsections. Section 3.5.1 discusses results relative to the single-cluster architecture and all the data and metadata allocation variants discussed in Section 3.4.3. Section 3.5.2 shows results for the multi-cluster architecture when data distribution techniques (discussed in Section 3.4.3.1) are compared to standard OpenMP data placement and caching.

3.5.1 Data Sharing and Distribution in a Single Cluster

This section focuses on an architectural template consisting of a single cluster (see Section 3.3) and explore the effect of placing shared data and support metadata onto different memory modules in the intra-cluster hierarchy. Here it's shown how efficiently implementing compiler and runtime support to data sharing through ad-hoc exploitation of the memory hierarchy is key to achieving performance and to overcome scaling bottlenecks.

The benchmarks are run under the allocation combinations previously described in Section 3.4.3:

- **Mode 1:** The default OpenMP placement. Data and metadata live in the master thread stack, which physically resides in the master core local L2 memory segment. Slave cores access them from there. This configuration is considered as a baseline for the experiments.
- **Mode 2:** Shared data resides in the master core local L2 memory. Metadata is replicated and transferred onto each core L1 SPM by means of a DMA transfer upon entrance into the parallel region. This mode reduces contention on master core L2 memory.
- **Mode 3:** Shared data is allocated in the non-cacheable segment of the shared L2 memory. Metadata resides on the master core local L2 memory. This mode reduces contention on the master L2 memory significantly.
- **Mode 4:** Shared data is allocated in the non-cacheable segment of the shared L2 memory. Metadata is replicated onto every L1 SPM. This configuration reduces the number of accesses to the master core L2 memory for data sharing to a minimum.

In case a program is memory-bound and most accesses are performed on shared arrays, high-contention on a single memory bank is bound to occur. In this situation, as discussed in Section 3.4.3, splitting arrays and allocating each partition on a different memory block can mitigate the effect of request serialization on a single memory device port. To investigate the effect of this kind of contention, two additional placement variants are considered, that leverage the data cache to implement array partitioning:

- **Mode 5:** Equivalent to mode 3, but shared data is placed in the cacheable segment of the shared L2 memory.
- **Mode 6:** Equivalent to mode 4, but shared data is placed in the cacheable segment of the shared L2 memory.

They are summarized in Table 3.2.

	Shared data	Metadata
Mode 1	Master core local L2	Master core local L2
Mode 2	Master core local L2	Local L1 SPM
Mode 3	Non-cacheable Shared L2	Master core local L2
Mode 4	Non-cacheable Shared L2	Local L1 SPM
Mode 5	Cacheable Shared L2	Master core local L2
Mode 6	Cacheable Shared L2	Local L1 SPM

Table 3.2: Shared data and metadata allocation variants

The benchmarks were run under each of the described modes. This set of experiments adopt the barrier that employ the distributed Master-Slave algorithm. Results of this exploration are reported in Figure 3.13. The curves there plotted show the scaling of the execution time speedup with the number of cores. Speedup results are normalized to the run time of the baseline allocation **Mode 1** (the default OpenMP placement) on a single core.

In general the various allocation modes allow increasing degrees of improvement with respect to default placement **Mode 1**, with the exception of benchmark *Pi Computation*, which shows no difference between modes. *Pi Computation* computes π by means of numerical integration. All threads participate in a parallel reduction loop. The reduction operation is implemented in such a way that all processors accumulate partial results onto a private variable, which is physically mapped onto each core local L2 memory. No contention arises during this operation. At the end of the parallel loop every processor atomically updates the shared variable by adding its partial result. Since the critical section has a very brief duration with respect to loop execution, changing the allocation of the shared variable does not show significant performance improvements.

Focusing on the rest of the benchmarks, it can be seen that replicating metadata onto every core L1 SPM (**Mode 2**) allows significant improvements with any number of cores. For processor counts up to 8, this mode is on average faster than simply allocating shared data in non-cacheable shared L2 memory (**Mode 3**), and slightly slower than accessing metadata from local L1 SPMs and shared data from non-cacheable shared L2 memory (**Mode 4**). This suggests that for most benchmarks the interconnect medium is congested when both metadata and data are accessed from the master core local L2

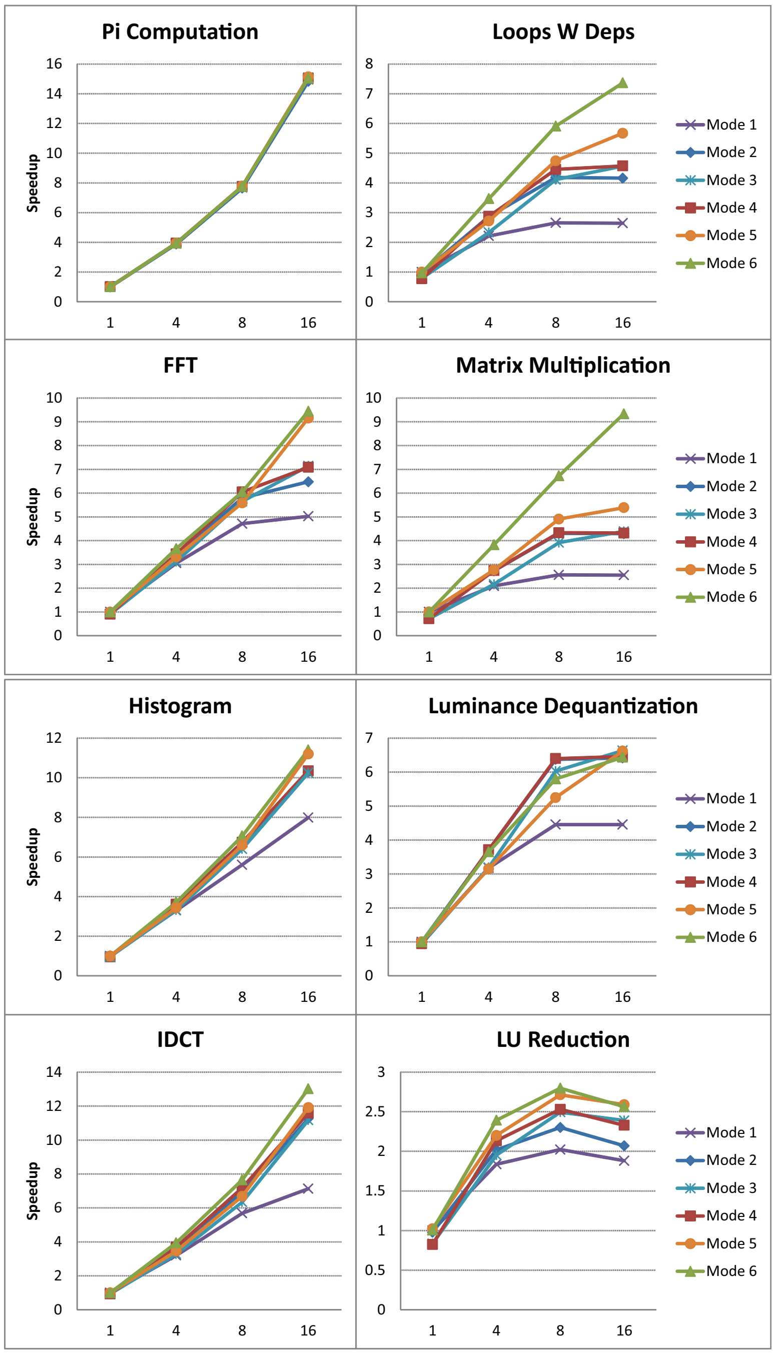
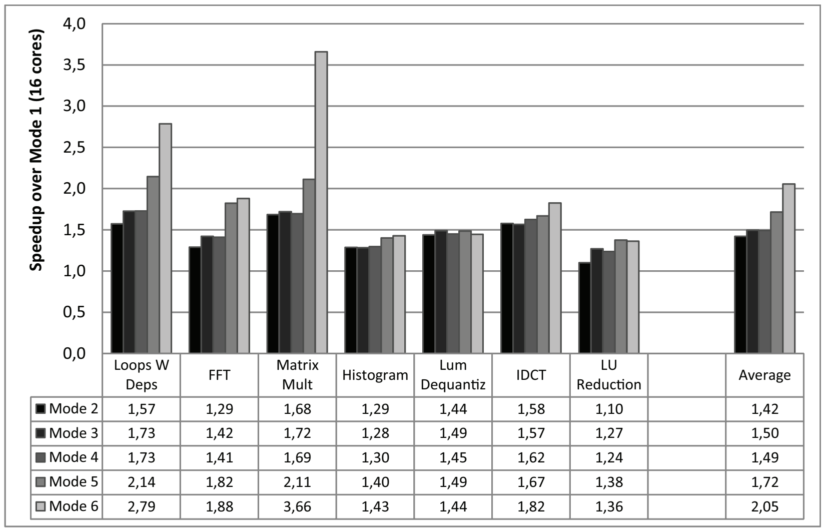


Figure 3.13: Scaling of different allocation strategies for data sharing support structures.

memory, but it is sufficient to divert the traffic towards one of the two items onto a different memory bank to offload the network.

For 16 processors the behavior changes slightly, and in many cases mode 4 performs identically to modes 2 and 3, particularly for the benchmarks *Loops W Deps*, *Luminance Dequantization* and *Matrix Multiplication*. Figure 3.14 shows the speedup of **Modes**



	Loops W Deps	FFT	Matrix Mult	Histogram	Lum Dequantiz	IDCT	LU Reduction		Average
■ Mode 2	1,57	1,29	1,68	1,29	1,44	1,58	1,10		1,42
■ Mode 3	1,73	1,42	1,72	1,28	1,49	1,57	1,27		1,50
■ Mode 4	1,73	1,41	1,69	1,30	1,45	1,62	1,24		1,49
■ Mode 5	2,14	1,82	2,11	1,40	1,49	1,67	1,38		1,72
□ Mode 6	2,79	1,88	3,66	1,43	1,44	1,82	1,36		2,05

Figure 3.14: Speedup of several data sharing support variants against the baseline for 16 cores.

2-6 against **Mode 1** for 16 cores only. This plot shows that on average **Mode 2** results in approximately 1.42x speedup. **Mode 3** achieves approximately 1.50x speedup, but **Mode** 4 does not do any better. This behavior is due to the above mentioned effect of serialization of accesses on the port of the memory device hosting shared data. As expected, allowing the cache to distribute shared data among different memory banks solves the problem and achieves excellent scaling. Partitioning shared data also magnifies the benefits of diverting metadata and/or shared data traffic out of the master core local L2 memory (**Modes 3** and **4** vs. **Modes 5** and **6**).

LU decomposition shows the worst scaling performance, only allowing a peak $2,8\times$ speedup for 8 cores and worsening for 16 cores because of the parallelization scheme and the dataset size. The algorithm operates on 32×32 matrices, which are scanned – with an upper-triangular pattern – within a nested loop, the innermost loop being parallelized. More precisely, the outer loop scans matrix rows. Row elements are operated on in parallel within the innermost loop. Since the number of row elements become smaller as the row index increases, at some point there will be more processors than elements to process. From this point of the computation on, an increasing number of processors will be idle (up to $N-1$ in the last iteration). This ”point” is obviously reached earlier for larger core counts, thus explaining the performance degradation from 8 to 16 cores.

3.5.2 Data Sharing and Distribution in the Multi-Cluster Platform

This section shows a new set of results aimed at evaluating the importance and effectiveness of the proposed distribution techniques to achieve a scalable execution of OpenMP applications on a multi-cluster MPSoC. The focus is on array-based applications between those presented previously, and the custom `distributed` directive citemarongiu is combined with the standard OpenMP loop parallelization `schedule` clause to achieve an efficient array partitioning scheme.

As previously, the baseline for experiments is the standard OpenMP policy for shared data placement, still called **Mode 1**. In **Mode 1** all shared data resides on the private (local L2) memory of the master core. Regarding metadata placement, we will always refer to the most efficient scheme among those discusses in the previous section. Throughout the rest of the section metadata is thus always assumed to be replicated in each processor's L1 SPM. To maintain the names (and meaning) of the modes introduced previously, only odd-numbered are considered among previous modes. As such, **Mode 3** and **Mode 5** allocate shared data in the uncacheable and cacheable regions of the shared L2 memory of the cluster 0, respectively.

New **Mode 7** and **Mode 8** are introduced, where shared data is distributed among uncacheable and cacheable segments of the shared L2 segments of clusters, respectively. Modes are summarized in Table 3.3.

	Shared data
Mode 1	Master core's local L2
Mode 3	Non-cacheable Shared MEM *(on cluster 0)*
Mode 5	Cacheable Shared MEM *(on cluster 0)*
Mode 7	Non-cacheable Distributed Shared MEM
Mode 8	Cacheable Distributed Shared MEM

Table 3.3: Shared data and metadata allocation variants in the clustered system

Figure 3.15 shows the results for these experiments. For the two kernels LD and IDCT (both coming from a JPEG decoding process), Mode 1 and Mode 3 perform almost identically poorly because both kernels are memory bound and thus performance is influenced by the bottleneck on the unique shared memory device being concurrently accessed by an increasing number of cores. This is confirmed by the fact that allowing shared data to be cached (Mode 5) execution results in nearly 2x speedup. The use of the cache, however, does not improve scalability, due to frequent misses traveling through the NoC induced by a round robin distribution of the fine-grained workload. Data distribution (Mode 7) results in perfect affinity between threads and memory tiles for these benchmarks, as is confirmed by its good scalability. Allowing shared data in cache from different shared memories further improves performance, since data is accessed from faster L1 memories. Data distribution dramatically improves caching benefits since every miss is serviced from within the cluster, and no miss traffic is injected in the NoC.

For FFT all the modes scale equally well since the benchmark is computation-intensive (power, exponential and logarithm). Due to the loose memory dependency compared to CPU it is thus impossible to appreciate the benefits of data distribution over standard data sharing

The histogram benchmark is representative of a class of irregular applications that feature subscripted access patters. While the work on the target image can be precisely divided among processors, it is impossible to foretell the access pattern on the histogram itself. This benchmark processes an input image consisting of randomly generated pixels, which implies that the memory access pattern is irregular and totally unpredictable. This clearly goes against the principle of data distribution, which assumes that an affinity between memory and threads can be statically enforced. Mode 7 is thus unfavored. This is in part confirmed by the results, which show that only modes 1 and 3 perform worse, while caching (model 5 and 8) improves performance.

Similar considerations are true for the LU reduction kernel as well. Moreover, from the previous paragraph it is known that the parallelization scheme adopted in the algorithm is inherently non scalable, since parallelization takes place on rows, whose items are increasingly fewer as an upper-triangular matrix is scanned, thus leading to a large amount of idleness on most processors.

Finally, matrix multiplication is another data-parallel algorithm, for which is easy to see the benefits introduced by data partitioning. Since it is data-intensive, Mode 1 unsurprisingly shows the worst results, which can in part be improved by modes 3 and 5. However, no scalability is achieved with these approaches, for the same reasons explained for the LD and IDCT kernels. Data distribution significantly further improves the general performance, and allows parallelization to scale to some degree. No better results could be achieved in this particular implementation of the matrix multiplication because one of the input matrices is integrally read by every thread, thus making void the effect of distribution on that matrix.

3.6 Conclusion

Software development in the embedded MPSoC domain is becoming increasingly complex as more and more feature-rich hardware is being designed. Pioneers pointed out the challenges in porting OpenMP to complex MPSoCs, where the compiler and runtime support must be revisited to account for the peculiarities of heterogeneous hardware (memory and/or computing resources).

Previous research in this field either is specific to a platform, or lacks a detailed analysis of performance implications of OpenMP programming patterns on the underlying hardware. This chapter showed an OpenMP implementation for a multi-cluster embedded MPSoC based on a modified GCC 4.3.2 compiler and on a custom runtime library. A thorough study of the performance achieved by several implementative variants of synchronization and data sharing support is also presented, focusing both at the single cluster level and at the system-wide level. It demonstrates that careful implementation of such a support on top of a heterogeneous communication medium and NUMA memory are key to performance. An implementation of support for data distribution in a clustered MPSoC is also presented. Results confirm that data-intensive application significantly benefit from data distribution.

Ongoing and future work is focused on further extending the OpenMP standard with features to expose hardware features at the application level to a higher degree. This includes directives for improved shared data distribution, DMA transfers, task priority and exploitation of acceleration hardware.

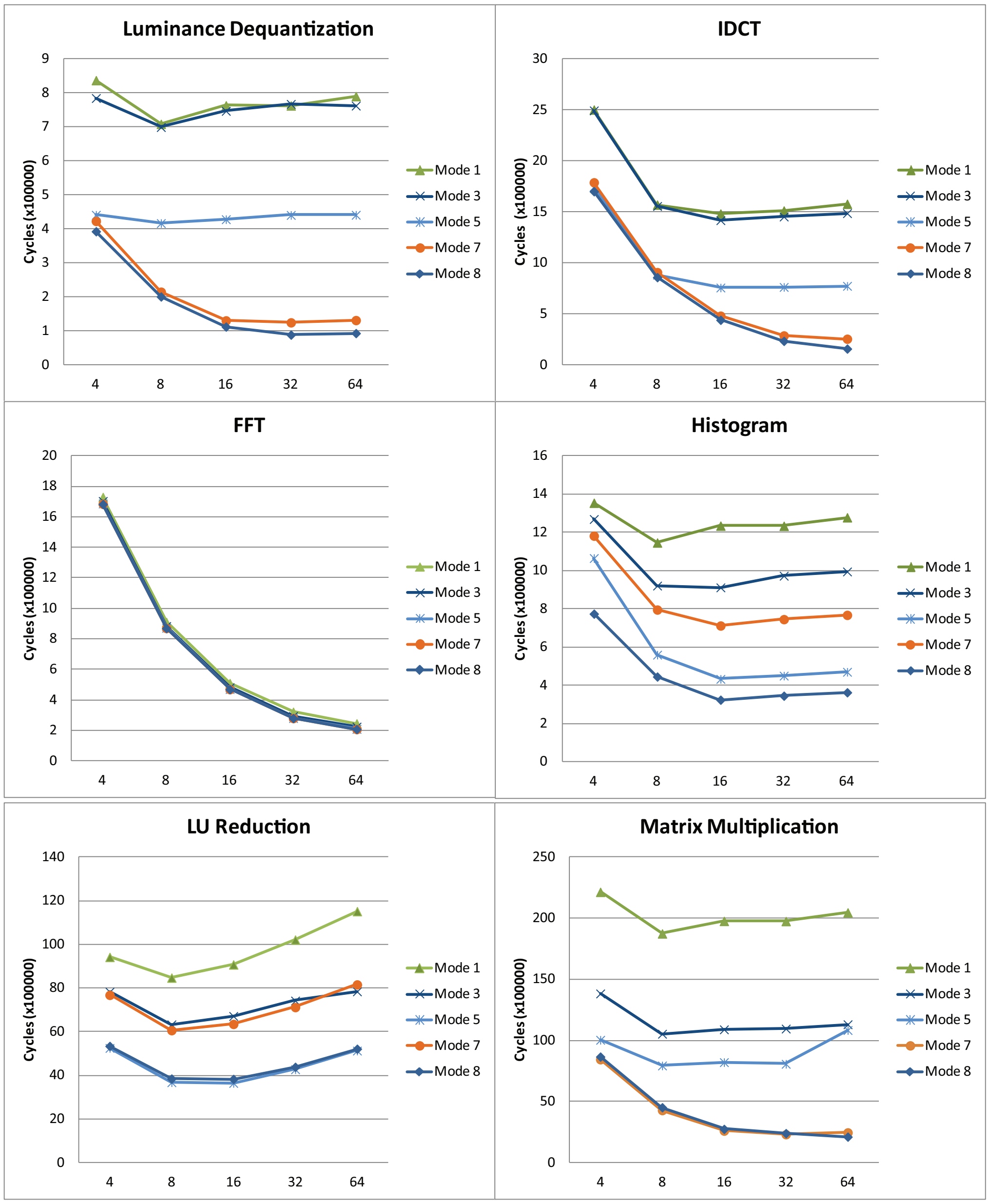


Figure 3.15: Comparison of performance (execution cycles) for several data sharing support variants in the multi-cluster platform

4

Workload distribution on 3D MPSoCs with shared memory

Modern applications expose execution pattern based on units of work (aka *tasks*) that are spawn and executed on parallel processing units. In a system where cores are arranged in tiles, and memory banks are physically partitioned (NUMA) among them, increasing the locality of task to the working data set is crucial to achieving performance. This chapter proposes two efficient workload distribution strategies for a shared memory MPSoC, and as a use case targets applications whose tasks are created out of iterations of a loop. The target platform has a silicon layer of multiple cores (up to 16 cores), and a shared DRAM whose banks are partitioned and 3D-stacked on top of the single tiles. To obtain high locality and balanced workload a two-step approach is considered. First, a compiler pass analyzes memory references in a loop and schedules each iteration to the processor owning the most frequently accessed data. Second, if locality-aware loop parallelization has generated unbalanced workload, idle processors are allowed to execute part of the remaining work from neighbors by implementing runtime support for work stealing. The two functionalities are exposed to application layer using a few simple extensions of the standard OpenMP frontend.

4.1 Introduction

Modern shared-memory systems are hierarchical, NUMA systems, with fast and low-power SRAMs tightly-coupled to cores, supported by bigger, yet slower and powerhungry DRAMs, typically placed off-chip. DRAM is accessed through on-chip I/O controllers which exploit sophisticated addressing mechanisms, thus making corresponding accesses slow and energy-hungry. Furthermore, DRAM controllers are shared among processors, which encounters scaling limitations when the complexity of the system increases.

Three-dimensional (3D) stacking technology provides a number of means to overcome the scalability limitations imposed on many-core integrated platform designs as 2D technology reaches the nanometer scale, both in general purpose and embedded computing (47, 76, 138). Traditional design constraints based on the evidence that the processor and memory subsystems had to be placed side by side can be overcome in 3D stacking (6, 49, 93, 138), where they can be placed on top of each other and linked through vertical interconnects based on Through-Silicon Via (TSV) technology which are more than two orders of magnitude more energy-efficient and denser than the most advanced off-chip I/O channels. Focusing on the high-end embedded domain, this ground breaking technology will enable the construction of multi and many-core data-processing systems with low latency and high bandwidth access to multiple, large DRAM banks in close spatial proximity.

This chapter considers a 3D-stacked platform for multi-dimensional array processing, which, for instance, targets image processing and computer vision domains. It features one silicon layer containing multiple processors organized in a two-dimensional mesh structure (communicating through a Network on Chip), and one or more DRAM layers containing the entire memory subsystem on the top. The memory space is shared among the cores, and explicitly software-managed, and adheres to the Partitioned Global Address Space (PGAS) paradigm. In this memory model each processor has quasi-ideal access to a vertical stack of memory banks in close vertical proximity. Memory transactions towards remote stacks travel through a horizontal on-chip interconnect (NoC), and are thus subject to an increasing cost with distance. The considered problem is to efficiently partition both workload (to cores) and to place data (in 3D-DRAMS) to maximize the accesses to local data.

The focus is on array-intensive applications, structured as a set of *doall* (i.e., data-parallel) loops, whose iterations can be independently distributed among processors. A naive assignment of iterations to processors, namely one which is unaware of the architectural assumptions and/or of the task and data mapping, leads to poor locality of memory references and/or load imbalance at runtime. In contrast to frequent cache- or DMA-initiated data transfers to improve locality, the presented approach schedules the workload (that is, loop iterations) in a locality-aware manner instead. Shared array structures in a target program are divided in as many tiles as processors, and tiles are distributed among their vertical DRAM stacks. A data-layout aware compiler analysis pass inspects to determine which particular tile (i.e. which memory) is being mostly referenced at each iteration. The iteration is assigned to the processor hosting the tile. The compiler statically inserts in the program the definition of local queues to each processor containing the description of work with high locality. Loops are restructured in such a way that at each iteration the work is fetched from these queues. The analysis pass requires that the access pattern performed on arrays is a statically analyzable affine function of the loop iterator. If this is not the case the analysis fails. However, profiling-based locality-aware parallelization is still allowed in this situation. The compiler instruments the program so as to gather access pattern information during a profile run. Profiling information enables the creation of high-locality work descriptors (queues) in case of an irregular application. As explained above, shared arrays are regularly distributed among memories. If the access pattern is not regular, or the loop iteration space does not overlap with the data space, it is possible that a subset of the tiles is accessed more frequently than the rest. This ultimately leads to assigning more iterations to the processor(s) holding these tiles. Stated another way, the queues may contain non-uniform amounts of work among processors, thus leading to unbalanced execution time. Here, *work-stealing* is used to mitigate this effect. Idle processors are allowed to steal part of the remaining work from remote queues in a locality-aware manner, thus achieving balanced execution and locality of references. The proposed techniques are compared against traditional data distribution or dynamic scheduling policies.

4.2 Related works

Recently, several 3D memory designs have been announced, confirming the benefits of 3D technology for high-efficiency next-generation memory systems (71, 75). Kgil et al. (76) present a high performance server architecture where DRAM is stacked on a multicore processor chip. Overall power improvements of 2-3× with respect to a 2D multi-core architecture are reported. Similarly, in (93) Loh presents a 3D stacked memory architecture for CMPs. By changing the internal DRAM architecture the author claims a 75% speedup. Industry leaders IBM and Intel are active in technology and architecture exploration (21, 47). Li et. al investigate in (87) the challenges for L2 design and management in 3D chip multiprocessors. Their term of comparison is 2D NUCA (Non-Uniform Cache Access) systems, which employ dynamic data migration to place more frequently-accessed data in the cache banks closer to the processor. Experiments show that a 3D L2 memory design with no dynamic data migration generates better performance than a 2D architecture that employs data migration.

3D memory integration is also actively explored in the embedded computing domain. All major players in the mobile wireless platform markets are very actively looking into how to integrate memories on top of MPSoC platforms for next-generation hand-held terminals (58).

More in general, the system size reduction, coupled with orders-of-magnitude improvements in memory interface energy efficiency are key enablers for disruptive innovation in embedded computing (49), possibly even more than in performance-centric general-purpose computing. In (108), Ozturk et al. explore core and memory blocks placement in a 3D architecture with the goal of minimizing data access costs under temperature constraints. Using integer linear programming, the best 2D placement vs the best 3D placement are compared. Experiments with single- and multi-core systems show that the 3D placement generates much better results (in terms of data access costs) under the same temperature bounds.

Concerning the parallelization techniques shown in this chapter, useful background work regarding the implementation of stealing policies can be found in (3, 131) and (31), whereas related research on data distribution is presented in (19, 32, 34) and (113). The main similarities are to be found in the language and programming model abstractions. Indeed, several patterns proposed in the past to efficiently program NUMA machines can be successfully adopted in the context of 3D MPSoCs. From the implementative point of view, however, the radical architectural differences between these machines require an in-depth reassessment of such techniques, based on the availability of a completely different hardware and software support for their construction.

A two-step approach to efficient loop parallelization on cache-based machines is proposed by Xue et al. in (143). Similarly to the technique presented here, they leverage static compiler analysis to schedule iteration in a locality-aware manner and runtime support for load balancing.

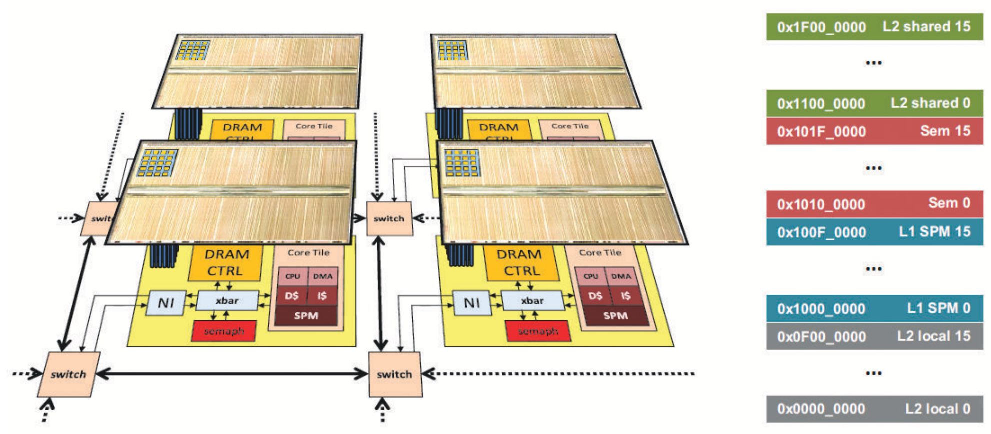


Figure 4.1: Target 3D architecture and PGAS

4.3 Target architecture and memory model

The platform template targeted by this work is the 3D-stacked MPSoC depicted in Figure 4.1. The bottom layer hosts the 2D multicore subsystem, whereas the topmost layer(s) consist of DRAM memory banks (93). Processing elements (PE) on the multi-core die feature a core tile, composed by a RISC-like CPU, a small amount of local L1 memory (SPM, caches) and a DMA engine. Each PE also hosts a set of local hardware semaphores implemented as a bank of registers with test-and-set read semantics and a fast DRAM controller with TSV DRAM physical interface for vertical communication to upper layers. Transactions towards remote memory neighborhoods are routed out of the PE by a Network Interface (NI), which injects them through the on-layer network (NoC) for horizontal communication. All the described IPs are interconnected through a crossbar, which is also in charge of determining whether memory references issued locally are to be transported vertically or towards the outer world. The memory subsystem leverages a Partitioned Global Address Space (PGAS) organization, and is thus accessible from the bottom layer by every tile through the described heterogeneous 3D interconnection. All of the on-chip memory modules are mapped in the address space of the processors, globally visible within a single shared memory space, as shown in the rightmost part of Figure 4.1. Despite this unique view of the memory space, each PE has a certain amount of tightly coupled physical memory, which we refer to as the processors memory neighborhood, and that is organized as a two-level hierarchy. L1 memory within each PE features separate instruction and data caches, plus scratchpad memory (SPM). Moreover, each PE is logically associated to a vertical stack of local L2 DRAM memory. The latter is logically organized in two parts. A shared segment (which constitutes part of the global shared memory), plus a (conceptually) private segment, where by default program code and private data to the core are allocated. L1 caches in this template are non-coherent, as hardware cache-coherence protocols are very expensive in terms of area, and scale poorly. To prevent inconsistencies, only private data and code to each processor are allowed to be cached. The logically private segment on each memory neighborhood is the only one that can by default be cached. Shared segments can only be directly accessed through the processor or DMA. It is a programmers responsibility to deal with coherency issues in case multiple copies of shared data are allowed. Similarly, if shared data is allowed to be cached, appropriate actions (e.g. flushes) must be taken in software.

4.4 Vertical Stealing

The target 3D MPSoC leverages a Partitioned Global Address Space (PGAS) organization of the memory subsystem, which has some affinities with traditional *Cache Coherent Non Uniform Memory Access* (CC-NUMA) multiprocessors (e.g. the SGI Origin (116)). Such machines typically contain a large number of processing nodes each with one or more processors and a portion of main memory connected through a scalable interconnection network. Although global memory is uniformly accessible by all the processors, remote memory latencies are typically much larger than local memory latencies. To obtain high performance on CC-NUMA machines is often necessary to distribute the data structures in the program so as to maximize the number of cache misses of each processor that are satisfied from local rather than remote memory. *Data distribution* (a.k.a. *array partitioning*) splits main arrays in the program in a set of tiles, which can be independently mapped on different memory nodes. Language abstractions and compiler techniques to enable data distribution in a program have been proposed in the past (19, 32) for CC-NUMA multiprocessors. Two common array distribution strategies are *block* and *cyclic*. *Block* distribution splits arrays in as many tiles as nodes thus assigning equally-sized tiles to each processor. *Cyclic* distribution allows the programmer to specify a partitioning granularity (i.e. a tile size). In both cases tiles are dealt out to processors in a round robin fashion. We describe a possible implementation of such facilities in Section 4.4.1.

To achieve high data locality it is important that a thread running on a processor operates on data which is hosted on the local memory neighborhood. *Block* (or *cyclic*) data distribution delivers good locality in case of regular loops. Indeed, the iterations of such loops can be distributed among processors in *chunks* whose size matches the array partitioning granularity. An example of such a scenario is provided in the code snippet below.

```
#define SIZE            16

/* The array A is block-distributed */
int A[SIZE];
#pragma omp distributed (A)

int i;

#pragma omp parallel for schedule (static)
for (i=0; i<SIZE; i++)
  A[i] = ...
```

Let us consider a target architecture composed by 4 processors. In the example above the array `A` is *block* distributed in 4 tiles of 4 elements each among the available *memory neighborhoods*. The array `A` is indexed with the loop induction variable `i`. This regular access pattern is amenable to static loop parallelization, where consecutive iterations are folded in chunks of 4 and assigned to processors in a round robin fashion. In this simple example there is perfect affinity between each thread and the referenced dataset.

When more complicated access patterns are executed *block* distribution fails in delivering good locality. To solve this issue arrays should be re-distributed, in an attempt to match the array access pattern exhibited by the running thread. To re-distribute arrays we adopt DMA transfers, which update the content of each memory neighborhood. However, this solution suffers from two main issues. First, the array access pattern in a program may change frequently (e.g. across different loops). Trying to re-distribute arrays in memory accordingly may thus require high amounts of DMA transfers. Second, 3D technology enables big amounts of memory to be tightly coupled to PEs. As a consequence, large array tiles can be entirely hosted on *memory neighborhoods*.

Frequently moving such large data blocks is likely to compromise the benefits of improved locality. An alternative approach to moving data may be that of scheduling loop iterations to processors in a locality-aware manner. More specifically, it is possible to leverage compiler analysis of array accesses in a loop to determine which physical memory is mostly accessed at a given iteration. The iteration is then scheduled to the processor owning that memory. The proposed *locality-aware* parallelization technique performs such analysis and builds *work queues* containing high-locality tasks (i.e. iteration descriptors) for each processor. A detailed description of the technique is given in Section 4.4.2.

Locality-aware loop parallelization does not require to move array tiles. Given an initial (e.g., *block*) distribution, all loops are re-structured in such a way that each processor is assigned the iterations that insist primarily on the tiles hosted on the local memory. A clear drawback of this policy is that processors may be assigned a different number of iterations, thus possibly leading to load imbalance among parallel threads. Let us consider the following example.

```
#define ROWS            16
#define COLS            16

int pix[ROWS][COLS];
#pragma omp distributed (pix)

int i, j;

/* Loop to parallelize */
#pragma omp parallel for schedule (static)
for (i=7; i<ROWS; i++)
  for (i=7; i<COLS; i++)
    pix[i][j] = ...
```

The matrix `pix` is *block* distributed among four available memory neighborhoods. Each memory hosts a tile of 64 elements. Corresponding cores must be assigned the iterations of a loop which operates on a subset (lower loop boundaries are greater than zero) of the matrix. Figure 4.2 highlights the part of the array which is accessed in the loop, and the corresponding layout in memory. If locality of accesses drives paralleliza-

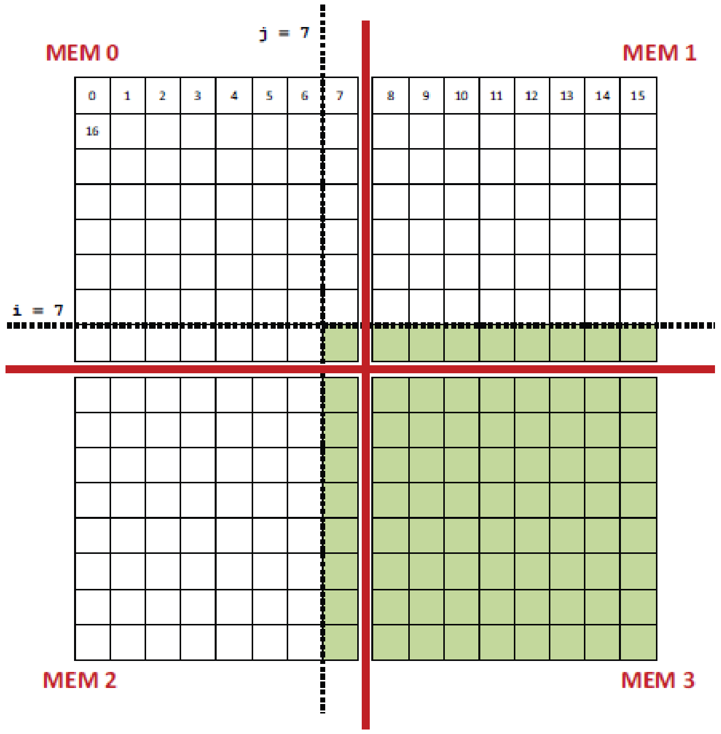


Figure 4.2: Layout of *blocked* array `pix` in memory and loop footprint on the array.

tion, a different number of iterations is assigned to each core. Figure 4.3 shows that only one element *from* tile 0 which is hosted on Memory neighborhood 0 (MEM 0) is accessed in the loop, thus processor 0 will be assigned a single loop iteration. On the contrary, all elements belonging to tile 3 hosted on Memory neighborhood 3 (MEM 3) are accessed in the loop. Processor 3 will be assigned 64 iterations, thus leading to load imbalance. Remote accesses on our MPSoC are subject to an increasing cost with the distance (i.e. the number of hops traversed in the NoC). However, different from CC-NUMA machines all the communication travels through tightly coupled layers, and thus fetching remote data on our MPSoC is much cheaper than an equivalent access on CC-NUMAs. Consequently, to solve the load imbalance issue we can afford the cost to allow idle processors to execute iterations originally assigned to other cores. Even if the stolen work has poor locality, the increased cost for remote references may still be repaid by load balancing. Section 4.4.3 presents the implementation of a runtime support to *work stealing*.

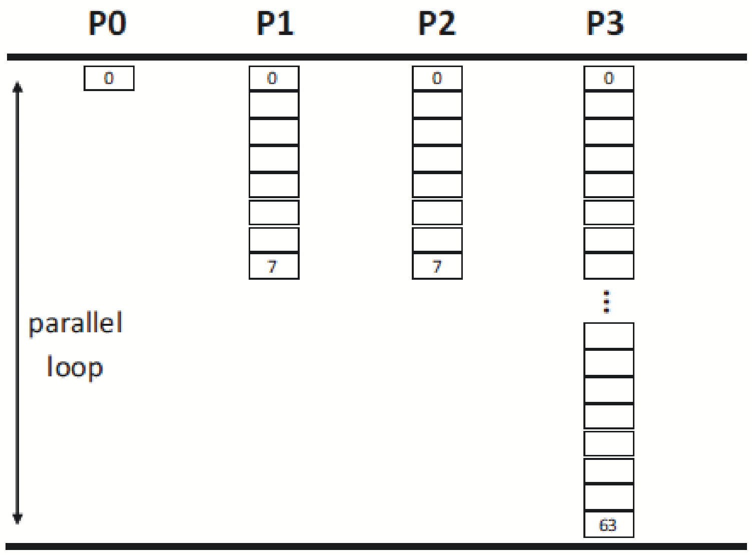


Figure 4.3: Imbalanced work queues.

4.4.1 Array Partitioning and Distribution

Programmer can trigger array partitioning with a custom `distributed` directive (94), as follows.

```
int A[1024];
#pragma omp distributed (A[, tilesize])
```

The `tilesize` parameter is used to specify the granularity of partitioning, namely the size expressed in terms of array elements of the elementary tile. The allocation policy follows a *cyclic* distribution scheme. *Block* distribution can be triggered by properly tuning the partitioning granularity so as to generate a number of tiles equal to the number of processors. This is automatically done by default if no `tilesize` parameter is given.

The primary concern when distributing data on CC-NUMA architectures is that physical placement of data must be performed in units of an operating system page, thus constraining the granularity of partitioning. If array tiles are much smaller than a page size data items to be places in local memories of distinct processors may lie within the same page. This situation leads to false sharing, and requires expensive data transfers within the virtual address space of the process to map different tiles to distinct pages. In alternative, data padding at the page level can be applied as a workaround, but this leads to significant memory wastage. The scenario is quite different for the target MPSoC. No specialized MMU hardware or OS support to virtual memory management is available, thus data partitioning is implemented in a lightweight manner by means of software address translation. Accesses to arrays annotated as `distributed` in the program are instrumented by our compiler with necessary instruction to locate the correct memory neighborhood at runtime as shown in Table 4.1. In this table `t` is the

Original reference	Transformed reference
`A[i]`	`(*tiles_A[i/t])[i%t]`

Table 4.1: Compiler instrumentation of distributed array accesses

size of a tile for the distributed array `A`, and `tiles_A` is a compiler-generated metadata array containing the base address for each tile of `A`. Indexing this array with a tile ID returns the base address for that tile. The ID of the tile being accessed is simply obtained by dividing the current offset (i.e. the array index) by the tilesize `t`. Once the base address of the target tile has been retrieved a modulus operation between the same operands returns the offset of the reference within the current tile.

The availability of such an efficient and streamlined implementation of the necessary support to data distribution enables to partition arrays at arbitrary granularities without wasting memory resources or incurring in data copy overheads. Furthermore, the cost (overhead) for a partitioned array reference does not change if different partitioning granularities are considered.

4.4.2 Locality-Aware Loop Partitioning

Data distribution schemes such as `block` or `cyclic` attempt to capture the most common array access patterns in loop-intensive applications. However, the access pattern may change at different points in a program (e.g. at different loops or parallel regions). For this reason, it is necessary that the programming model or the compiler allows to *re-distribute* arrays across different regions or to schedule loop iterations under a certain affinity with the current layout of array tiles in memory. Continuously re-distributing arrays may lead to a high number of data transfers. On the target platform, the amount of memory made available by 3D stacking allows each memory neighborhood to accommodate big-sized tiles, whose frequent movement is likely to significantly impact the performance. Affinity-based loop scheduling techniques appear therefore more suitable to address the described issues. An example of such a technique is the *Owner-Computes Rule* from the High-Performance Fortran (HPF) compilation system, which after distributing the ownership of array elements to the processors, distributes the charge of executing each instruction to the processor owning the variable modified by this instruction (i.e. the *Left-Hand Side* expression of an assignment statement). This may still lead to high amount of communication, since components of the *Right-Hand Side* expression may have to be communicated to the owning processor before the assignment is made.

A locality-aware parallelization strategy seems more appealing, because it assigns an iteration to the processor whose memory neighborhood hosts the most frequently referenced array tile(s) within that iteration.

To this aim, the GCC compiler was modified – as described in Section 4.4.2.1 – to include a static analysis pass. This analysis can be applied to counted loops whose array subscripts are affine functions of the loop iterator. In case the loop does not satisfy such requirements, locality-aware parallelization is still allowed by leveraging profile information (Section 4.4.2.2).

4.4.2.1 Static Analysis

The static component analysis operates on the following setup. Architectural information required is the number N of processors, to which corresponds a set of m associated *memory neighborhoods*.

$$m \in M = \{m_1, ..., m_n\} \tag{4.1}$$

Let L be the set of loops in a program, and D the set of distributed arrays. Each loop $l \in L$ has an associated iteration space I_l. Within the loop body executed at each iteration $i \in I_l$, a number $K \in \mathbb{N}$ of accesses to distributed arrays is performed. Every access can be characterized with a subscript function $s_{d,j}$, where $d \in D$ and $j \in \{1, .., K\}$. These subscripts must be an affine function of the loop iterator i, namely

$$s_{d,j}(i) = f(i) = a * i + b \tag{4.2}$$

where $a, b \in \mathbb{N}$. This ensures that the compiler can determine the exact offset at which the target array is accessed.

Remind from Section 4.4.1 that a distributed array declaration conveys to the compiler information about the partitioning granularity (i.e. the size of a tile). Based on this information, and on the offset described by the *subscript* function, every array access can be brought back to a specific tile, and finally to the memory hosting that tile. If t_{SIZE} is the size of a tile for the current *distributed* array, the ID t_{ID} of the tile being accessed can be determined as follows:

$$t_{ID} = \frac{s_{d,j}}{t_{SIZE}} \tag{4.3}$$

Since *cyclic* distribution is triggered by the compiler for `distributed` arrays, it is possible to determine on which memory neighborhood m_{ID} a given tile is mapped to

$$m_{ID} = t_{ID} \% N \tag{4.4}$$

In short, if we indicate with S the set of all the subscripts representing array accesses, we define a map function that associates each array access to a physical memory.

$$map : S \rightarrow M \tag{4.5}$$

Statements within the loop body are walked, and every array access found is analyzed as discussed. The outcome of this analysis step is a multiset M_i, which describes the cardinality $n(m)$ of each memory m accessed in the iteration i.

$$M_i = \{(m, n(m)) : m \in M\} \tag{4.6}$$

The memory with the highest cardinality is the one with the highest affinity to the current iteration, which is then assigned to the processor owning the memory. The described analysis has been implemented within the OpenMP expansion pass in the GCC 4.3 compiler (52), and is triggered by the use of the custom `locality` scheduling clause for the original OpenMP `#pragma omp for` loop parallelization directive.

```
#pragma omp for schedule (locality)
for(i=LB; i<UB; i+=step)
  /* Loop body */
  exec_loop_body (i);
```

At the end of the analysis a *queue* containing the description of work with high locality is created for each processor and for every loop. The original loop code is transformed as shown in the code snippet below. As will be explained in Section 4.4.3, the queues are managed through *head* and *tail* pointers which reside in each processor L1 SPM for fast inspection. At the beginning of a loop the corresponding queue is properly hooked to local pointers through the `omp_init_queues` function.

```
  int i, ii, has_work;
  omp_init_queues();

  has_work = omp_get_chunk (&lb , &ub);
  if (!has_work)
    goto LEAVE ;
LOOP:
        for (ii=lb; ii <ub; ii ++)
        {
                i = omp_get_iteration(ii );
                /* Loop body */
                exec_loop_body (i);
        }

        has_work = omp_get_chunk(&lb , &ub );
        if (has_work)
                goto LOOP;
LEAVE :
```

After each queue has been copied locally, the work is fetched from there with the `omp_get_chunk` function, which extracts part of the remaining work in the queue at each invocation. The size of the chunk of iterations extracted can be defined by the programmer. The range of queue elements to be processed upon this invocation is described as lower and upper bounds (`lb`, `ub`) of a compiler-generated loop with. The original loop iteration `i` is fetched from the *queue* through the `omp_get_iteration` function and passed to the loop body for execution.

4.4.2.2 Profile-based Analysis

To enlarge the scope of applicability of the proposed approach to benchmarks containing non-statically analyzable array accesses. The modified compiler can instrument the program so as to collect information about which memory is mostly referenced within each iteration during a *profile* run of the program. If the custom `-fomp-profile` flag is given, the compiler emits instructions that generate a trace of the array accesses.

These are captured at runtime by a script, which collects them into per-iteration access descriptors, and are passed to a *queue generator*. The *work queues* are described as a standard C array within a header file, which is later included for compilation during the second program run and linked to our enhanced OpenMP runtime library. The entire flow for the profile-based locality-aware parallelization is shown in Figure 4.4.

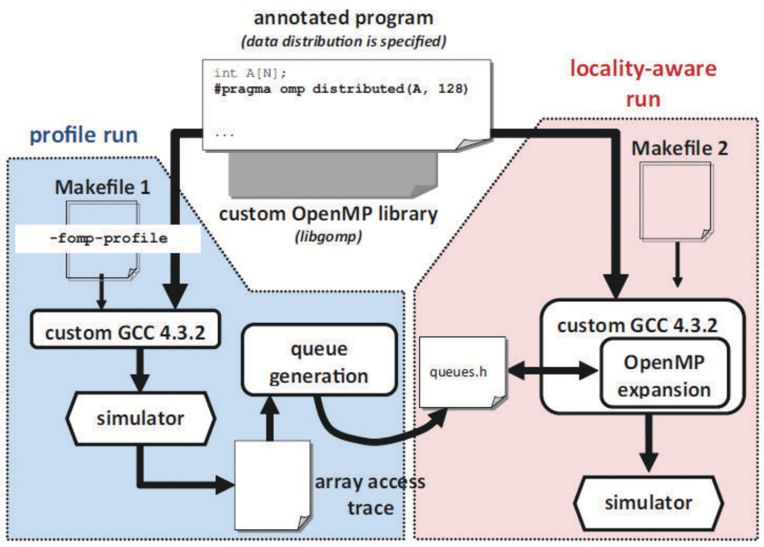


Figure 4.4: Profile-based analysis toolflows.

4.4.3 Runtime Support for Work Stealing

As introduced in the previous section, locality-aware loop parallelization is based on *work queues* that describe which iterations are assigned to each processor at a given loop. Every processor fetches its assigned iterations from these *queues* for execution in chunks. If an attempt to extract work from a local *queue* fails, meaning that either no iterations were assigned to the processor by the locality-aware parallelization pass or all pre-assigned iterations have been processed already, then a *work stealing* policy may be triggered so that an idle processor can transfer part of a remote *queue* to its local descriptor and continue working.

Programmer can enable this kind of scheduling by associating the custom `stealing` scheduling clause to an OpenMP loop, as follows.

```
#pragma omp for schedule (stealing[, range]*/)
for(i=LB; i<UB; i+=step)
  /* Loop body */
  exec_loop_body (i);
```

which gets transformed into:

```
  int i, ii;
  int has_local_work, has_global_work;

  omp_init_queues ();

  has_local_work = omp_get_local_chunk (&lb , &ub);

  if (!has_local_work)
    goto STEAL;
LOOP:
  for (ii =lb; ii <ub; ii ++)
  {
    i = omp_get_iteration (ii);
    /* Loop Body */
    exec_loop_body(i);
  }

  has_local_work = omp_get_local_chunk (&lb , &ub );
  if (has_local_work)
    goto LOOP;

STEAL :
  has_global_work = omp_steal (&lb , &ub);
  if (has_global_work)
    goto LOOP;
```

The compiler restructures the original loop as two nested loops, a *work* loop and a *steal* loop. Each processor owns two local queues, a Non-Stealable Queue (NSQ) containing the set of iterations being currently processed, and a Stealable Queue (SQ), visible to other stealers. From an implementative point of view this double-queue system leverages a single multi-indexed memory region (the *work queue*), where *head* and *tail* pointers to stealable/non-stealable elements are updated through lock-protected operations. The *work queue* resides on main (DRAM) local memory, whereas control pointers are allocated on SPMs for fast inspection, as shown in Figure 4.5.

Each processor attempts to fetch work from the local *queue* through the `omp_get_local_chunk` function, which sets lower and upper bounds for the *work* loop. The function returns the size of the extracted work chunk. In case the chunk size is zero, no local work is left to do, and thus a *steal* operation is attempted. Since array data are never moved,

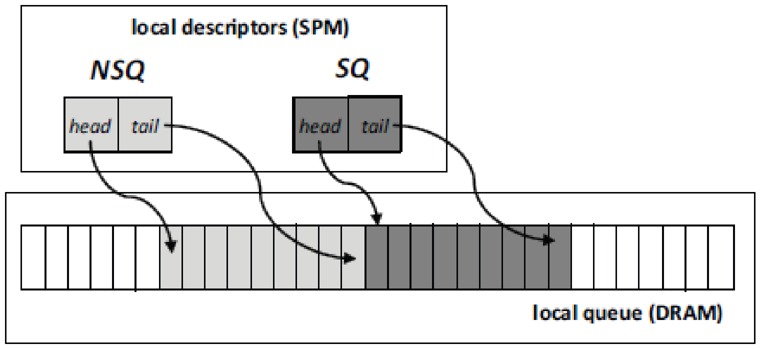


Figure 4.5: Implementation of the *work queues* and descriptors of stealable/non-stealable regions.

allowing a processor to steal work from other cores breaks the locality contained in the original work assignment. In case of *memory-bound* parallel loops allowing an idle processor to steal work from far away processors is likely to significantly degrade performance due to the high number of costly remote accesses. However, stealing only from nearby processors may still be beneficial. Based on this rationale, a stealing policy was implemented, in which the *stealer* can only fetch work from processors within a given distance. The programmer can annotate a maximum steal `range` (specified as the maximum allowed number of hops to look for stealable work) to the `schedule (stealing)` clause. The stealing policy is implemented within the `omp_steal` library function, shown in Listing 4.1.

Upon entrance into the function, each processor annotates in the shared variable `local_done` the information that it has no more work in its local *queue*. A *stealer* then continuously considers other processors as possible *victims*. First, the distance (in number of hops) between the *stealer* and the *victim* is inspected from within a lookup table (LUT). If the distance is out of the allowed `range` the *victim* is discarded, otherwise it is a good candidate for the *steal* operation, which is triggered by a call to the `omp_get_remote_chunk` function. The loop continues until every processor has entered at least once the `omp_sleep` function, thus signaling that no processor has local work left to perform.

```
int omp_steal  (int *lb, int * ub)
{
  /* Get processor ID */
  int pid = get_proc_num ();

  /* Prior to entering the steal loop notify
     that current processor has completed local work */
  local_done |= 1<<pid;

  while (1)
  {
    /* Iterate over processors */
    for (i =0; i< NUM_PROCS; i++)
    {
      /* No processor has local work left.
         Nothing to steal. Quit. */
      if (local_done == 0xffff)
        return 0;

      /* Determine distance between stealer and victim */
      int distance = LUT (i, pid);

      /* Current victim is farther than
         maximum allowed steal distance */
      if (distance > range || !distance)
        continue;

      int chunk = omp_get_remote_chunk (i, lb , ub);
      if (chunk)
      return chunk ;
    }
  }
  return -1;
}
```

Listing 4.1: `omp_steal` runtime function

4.5 Experimental results

This section describes the experimental setup used to evaluate the proposed programming framework, and the results obtained. The OpenMP-based programming framework with the proposed extensions was implemented within the GCC 4.3 compiler (GOMP (52)). The runtime environment (`libgomp`) adopts a MPSoC-specific implementation (94) which does not leverage OS support nor thread libraries. Each OpenMP thread is pinned to a given processor based on its ID. The library code is executed by every core. At system startup the processor with the highest ID is designated as the master processor, and it is responsible for orchestrating parallel execution by synchronizing slave processors and pointing them to parallel code and shared data. An instance of the 3D platform template presented in Section 4.3 was implemented within a SystemC full system simulator (26). The simulated 3D chip is composed by three layers. The bottom level hosts 16 processor tiles, while L2 memory stacks (16 MB each) reside on the topmost two layers, respectively devoted to the shared and private segments. On-tile L1 memory features 16KB scratchpad memory (SPM) plus separate data (4KB) and instruction (8KB) caches. It is worth recalling here that caches only manage private data, therefore preventing any coherence issues. Figure 4.6 shows how PEs are placed on the CMP die. Processor IDs increase with the pattern indicated by the ar-

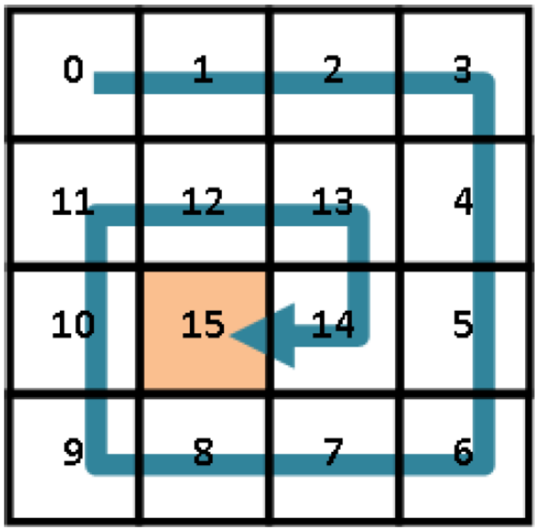


Figure 4.6: Processor layout on the CMP die.

row. Because of OpenMP's master-slave execution paradigm, the program starts as a single thread of execution. All data declared out of the scope of parallel constructs is by default allocated on the memory neighborhood of the master core. Therefore, slave cores will sometimes need to communicate through this memory stack. To minimize the effect of the NUMA latencies seen by different slaves, the master core is kept in a central position in the CMP die.

The memory access time depends on the transaction path. Accesses to local SPM are subject to only 1 cycle latency. For remote SPMs this cost depends on the internal memory interface latency (≈ 2 cycles), the number of hops to the target memory controller, the contention level on the network, the neighborhood interface latency (≈ 2 cycles), the neighborhood memory latency (1 cycle for SPM, ≈ 5 cycles for 3D stacked DRAM). The network on chip on the CMP die is based on the ST Microelectronic STBus protocol. The zero-load NoC latencies for remote accesses depend on the number of traversed hops, and are modeled as shown in Figure 4.7. L is a parameterizable value

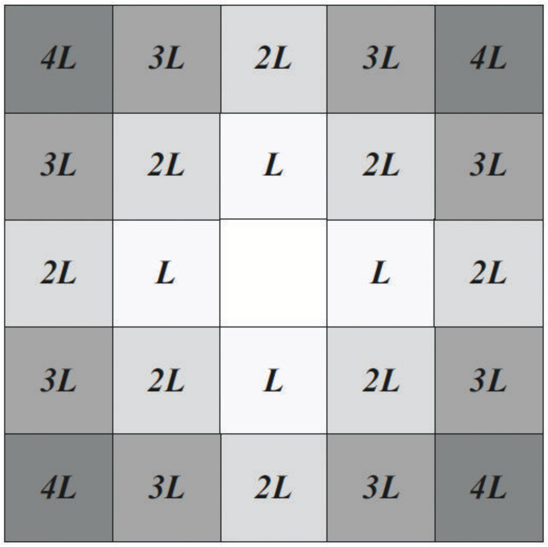


Figure 4.7: Zero-load NoC latency modeling.

which represents the cost to traverse a single hop. For example, if $L = 10$ in absence of contention accessing data on the memory neighborhoods of processors 4, 14 or 10 from processor 12 is subject to a latency of 20 cycles. If interconnect resources are shared with other concurrent transactions, the latency will be higher.

To test the effectiveness of the proposed techniques a synthetic benchmark (*Synth*) and 3 representative application kernels from image processing domain were considered, namely:

1. Inverse Discrete Cosine Transform (IDCT)

2. Luminance Dequantization

3. Matrix Multiplication

Each of these benchmarks is executed under the following program configurations:

- `static`: Static loop parallelization. An identical number of iterations is assigned to each processor
- `dynamic`: Dynamic loop parallelization. Work is scheduled in a *first-come first-served* fashion to processors in chunks of N (configurable) iterations
- `locality`: Locality-aware loop parallelization. The loop executes under the work description contained in the queues generated by the modified compiler.
- `stealing` (`range` = M): Locality-aware loop parallelization + *work stealing*. Processors that run out of work are allowed to steal some iterations from processors within a distance of M (configurable) hops.

Results of the experiments are collected in plots that show the execution time of each program run (in millions of cycles) for increasing values of the latency $L \in \{1, 5, 10, 15\}$.

4.5.1 IDCT and Luminance Dequantization

Results for IDCT and Luminance Dequantization (LD) kernels are shown in Figure 4.8. These are two kernels extracted from a JPEG decoder, which operate on an image composed by 600 DCT blocks. The two main differences between these kernels consist in the access pattern which is regular for LD and scattered for IDCT and in the number of accesses performed within each iteration, which is much bigger for IDCT (384 vs 64). Arrays are partitioned with *block* distribution, but since the number of memory neighborhoods does not evenly divide their size some processors own larger tiles than others. Similarly, the number of processors does not evenly divide the number of loop iterations (i.e. DCT blocks), and thus cannot capture with exact precision the affinity between iterations and tiles. For this reason, as L increases the performance of `static` scheduling decreases. Similarly, even if a certain (small) amount of unbalancing is present in this loop, `dynamic` scheduling is overwhelmed by the cost for remote references generated by this locality-agnostic parallelization scheme. On the contrary, the `locality` scheduling exactly establishes the affinity between an iteration and the corresponding array tile, as is confirmed by the fact that its execution time does not change for varying L. For a realistic value of $L = 10$, in the IDCT kernel `locality`

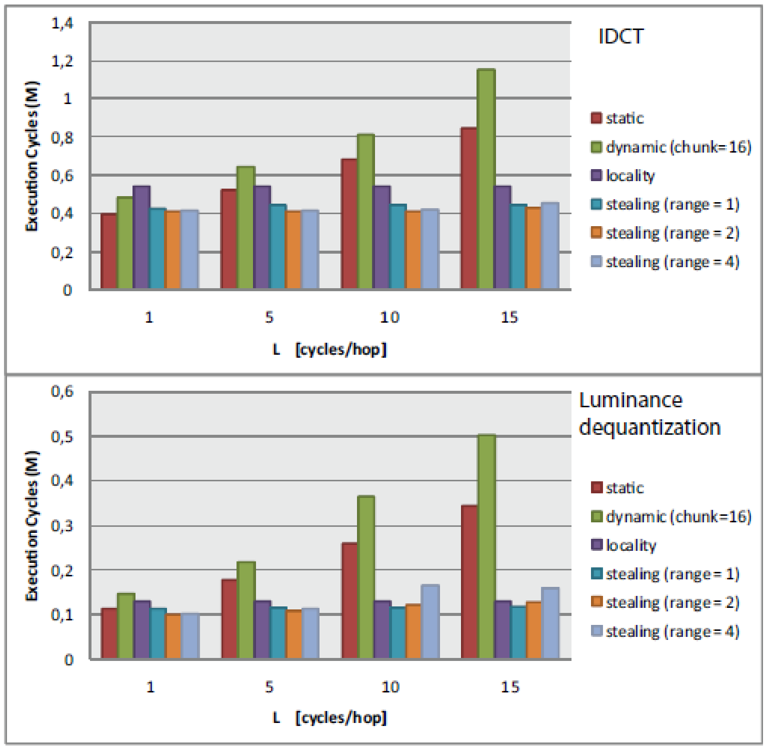


Figure 4.8: Results for IDCT and Luminance Dequantization.

scheduling is 21% faster than `static` scheduling, and *stealing* is up to 40% faster than `static` and 50% faster than `dynamic`. In the LD kernel `locality` is 50% faster than `static`, and `stealing` is up to 56% faster than `static` and 69% faster than `dynamic`.

4.5.2 Matrix Multiplication

Results for the Matrix Multiplication kernel are shown in Figure 4.9. This benchmark is amenable to static loop parallelization, which generates the iteration space partitioning shown in the plot on the left in Figure 4.10. *Block* distribution accommodates array tiles in memory so as to exactly match the threads footprint (see plot in the middle in Figure 4.10). A worst-case array distribution was forced for the `static` loop scheduling, such as the vertical blocking shown in the plot on the right in Figure 4.10. The plot in Figure 4.9 shows a very interesting result. As expected, *block* distribution associated to `static` scheduling delivers excellent performance because of the high amount of local accesses and the low scheduling overhead. For $L = 10$ it is possible to notice that our `locality`

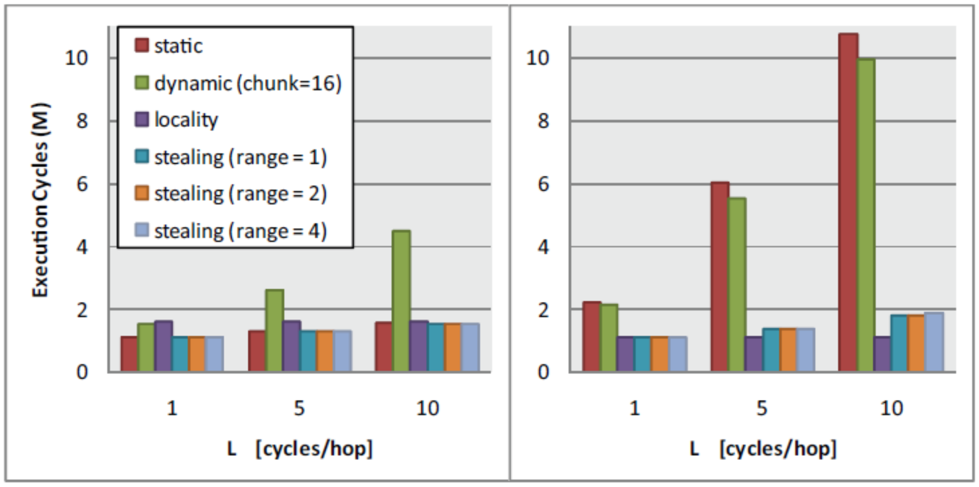


Figure 4.9: Results for Matrix Multiplication. Horizontal (left) and vertical (right) blocking.

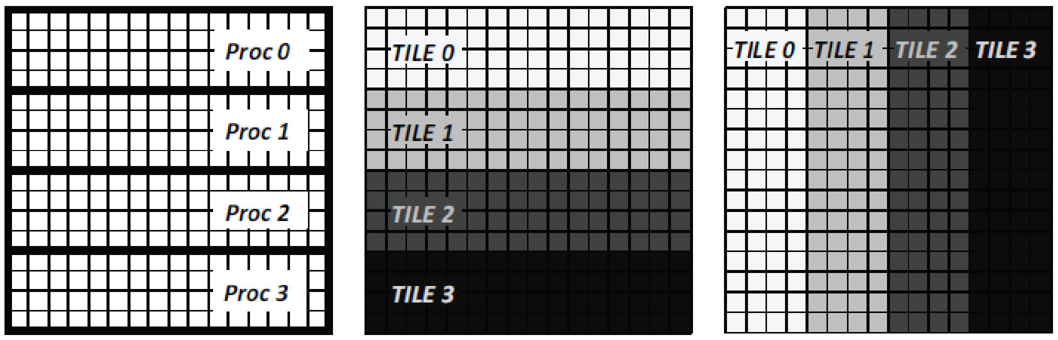


Figure 4.10: Static iteration space partitioning (left). Horizontal (middle) and vertical (right) blocking.

scheduling performs equally well. *Work stealing* can not do any better since the loop is highly balanced, but it does not degrade much the performance, thus indicating that the proposed techniques and runtime introduce a very low overhead. `dynamic` scheduling, which lacks any locality awareness, significantly degrades performance as L increases. When employing the unfavorable vertical *block* data distribution scheme, `static` scheduling worsens. It can be seen that the `locality` scheduling is insensitive to the data distribution scheme applied, thus delivering the best results. `stealing` scheduling does slightly worse, since as already pointed out the loop is well balanced, and thus dynamic techniques are not beneficial, and only add overhead.

4.5.3 Synthetic benchmark

The aim of this synthetic benchmark is that of forcing the generation of *work queues* which describe a very unbalanced loop scheduling (i.e. most iterations are assigned to a single processor), to study the effect of the `range` parameter of the stealing techniques. We will explore how the performance of work stealing changes when work is stolen from N-hop distant processors, with $N \in \{1, 2, 4, 6\}$.

In this benchmark an array of 16K elements is block distributed in 16 tiles of 1024 elements. A parallel loop with 1074 iterations accesses the first 1074 elements of the array, 1024 of which are contained in the first tile, and the remaining in the second tile. It it therefore clear that the locality-aware parallelization assigns 1024 iterations to the first processor, 50 to the second processor, and none to the other processors. Results for this experiment are shown in Figure 4.11. Unsurprisingly, the

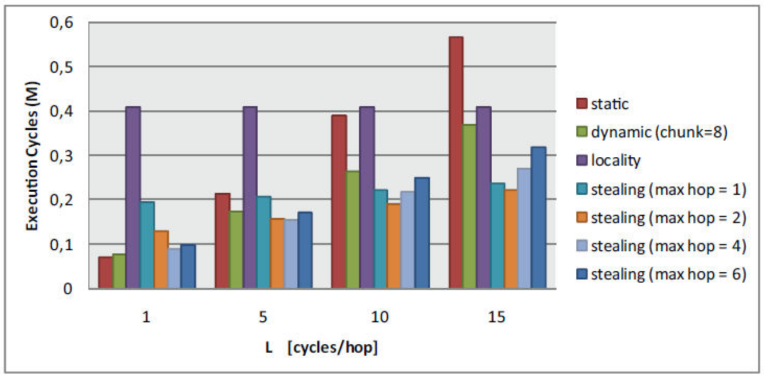


Figure 4.11: Results for the Synthetic benchmark.

`locality` scheduling performs poorly, since most processors are idle. When $L = 15$ the high cost for remote accesses renders `static` scheduling even slower. `dynamic` scheduling randomly assigns iterations to ready processors without caring about how costly the consequent communication will be. For this reason its performance degrades as L increases. `stealing` scheduling provides the best results, since it starts from an original mapping with high-locality, and then manages the imbalance by dynamically re-distributing the workload. It is possible to notice that the best results are achieved when `range` $= 2$. Recalling that all iterations are originally assigned to processors 0 and 1, and considering the position of these processors in the CMP layout (cfr. Figure 4.6), it is evident that for smaller values of `range` only two processors are allowed to steal from processor 0. On the other hand, when bigger values of `range` are allowed the cost for remote accesses dominates the benefits of `stealing`.

4.6 Conclusions

This chapter investigated the integration of a locality-based approach to loop parallelization with runtime support to work-stealing techniques as a convenient programming abstraction for 3D integrated embedded manycore platforms. The compilation strategy is based on a first analysis which associates an iteration to the processor which owns the referenced data. In case such analysis cannot be carried out at compile time, profile information are exploited to achieve the same result. At runtime idle processors are allowed to steal part of the remaining work from remote queues in a locality-aware manner, thus achieving balanced execution and locality of references. Results on a set of data-intensive kernels underlined the effectiveness of locality-aware parallelization. Work stealing appeared to be less beneficial on these benchmarks, where - however - computation among parallel threads is inherently balanced. Intuitively, stealing would allow much more significant speedups for imbalanced applications, which should therefore be considered in future work. Moreover, techniques were implemented to dynamically (i.e. at runtime) determining the affinity between a given iteration and a target memory neighborhood. These techniques can be adopted when a loop is not statically analyzable and profiling can not be exploited (i.e., applications whose execution flow is data dependant).

5

Support for nested and irregular parallelism on shared memory many-core clusters

Modern designs for embedded systems are increasingly embracing cluster-based architectures, that can deliver very high peak performance within a contained power envelope. However, making effective use of these platforms is becoming extremely difficult, as embedded applications are growing in complexity and express a parallelism that is less structured and regular than before. Parallel programming models have changed to cope with this, and so must do the runtimes that support them. This chapter focuses on two powerful abstractions, namely *nested* and *irregular* parallelism, and show the optimized design for a runtime for efficiently supporting them. OpenMP will be the target programming model, because of its great expressiveness, which allows us to extend the analysis and the proposed solutions also to other programming model.

5.1 Introduction

Modern embedded systems are embracing multi-clustered architectures, where each cluster is composed of a small-medium number (typically up to 16) of cores, interconnected through a high-bandwidth, low-latency communication and memory system, and inter-cluster communication is achieved through a scalable interconnection medium, such as a NoC. Its leverages a a shared memory model, in which each cluster can access local or remote (i.e., belonging to another cluster) L1 storage, as well as L2 or L3 memories. However, due to the hierarchical nature of the interconnection system, memory operations are subject to non-uniform accesses (NUMA), depending on the physical path that corresponding transactions traverse. Similar to traditional NUMA systems, nested (or multilevel) parallelism represents a powerful programming abstraction for these architectures. Exploiting a single level of parallelism means that there is a single thread (master) that produces work for other processors (slaves), and additional parallelism possibly encountered within the unique parallel region is ignored by the execution environment. When the number of processors in the system is very large, this approach may incur low performance returns, since there may be not enough coarse-grained parallelism in an application to keep all the processors busy. *Nested* parallelism is used to increase the efficiency of parallel applications in large systems, and implies the generation of work from different simultaneously executing threads, enabling better resource exploitation. In a cluster-based architecture nested parallelism is extremely beneficial, where a first level of parallelism can be used to distribute coarse grained tasks to clusters, and one or more inner levels of fine-grained (e.g., loop-level) parallelism can be distributed to processors within a cluster. This capability of confining a macro-tasks within the boundaries of a cluster is key to achieving locality and balancing, thus, performance. Nested parallelism can be implemented by using a mix of programming models, but this will make application development cumbersome, as the programmer is required to manually create threads and orchestrate their communication and synchronization using different paradigms. Moreover, this approach makes it difficult, if not impossible, the application of global policies (for instance, to perform load balancing or improve data locality) that cross the boundary of each layer. A more appealing solution is one where the programmer is allowed to create nested parallel regions from within a unique programming model, such as OpenMP. In this chapter, we will see the design for a runtime to efficiently support nested parallelism on shared-memory many cores cluster.

At the same time, embedded applications from the domains targeted by such architectures (e.g., image processing, computer vision, ...) are increasing in complexity and often expose high degree of parallelism which is irregular in nature and/or dynamically generated. The *tasking* execution model represents a powerful abstraction to exploit this kind of parallelism, as it enables asynchronous, dynamic creation of units of work in a simple and straightforward manner. However, the applicability of the approach is again limited to applications exhibiting units of work which are coarse-grained enough to amortize the overheads introduced by the support runtime. A second part of this chapter describes the design of an optimized runtime environment supporting the fine-grained tasks on an embedded shared-memory cluster. The key aspects critical to performance are identified, and several architectural support are proposed to minimize the effect of major bottlenecks implied by the execution model.

Finally, a hardware implementation of a generic Scheduling Engine (HWSE) which fits the semantics of OpenMP tasking is proposed. The adaptability of this HW block in the context of different programming models is also discussed. The HWSE is designed as a tightly-coupled block to the PEs within a multi-core cluster, communicating through a shared-memory interface. This allows very fast programming and synchronization with the controlling PEs, fundamental to achieving fast dynamic scheduling, and ultimately to enable fine-grained parallelism.

5.2 Related works

Nested parallelism can be implemented in different ways (11, 59, 73, 95, 121). In literature many techniques exist, which can be categorized into two main approaches:

1. **Dynamic thread creation (DTC)**: Whenever the application asks for additional parallelism, it is mapped on a lightweight thread from some standard package (e.g., *pthreads*). This approach allows very flexible creation of parallelism as needed, but has a major drawback: thread creation is expensive both in terms of space (memory footprint) and time (98), (40). In a resource-constrained platform such as the considered one, this approach would quickly run out of memory, and the resulting time overheads would disallow fine-grained parallelism.

2. **Fixed thread pool (FTP)**: A fixed number of lightweight threads (typically as many as the number of processors) is created at system startup and constitute a fixed pool of idle workers.

When a program requests the creation of parallelism, physical threads are fetched from the pool. If the number of logical threads created at an outermost parallel construct is less than the number of threads in the pool, some of them will be left unutilized and available for nested parallelism.

There also are many hybrid approaches, which combine in some ways DTC and FTP. Some techniques start with a FTP approach, and dynamically create new threads when there are no idle workers on the pool (52). Other solutions leverage thread creation at the outermost level of parallelism – where the computation is assumed to be coarse enough to amortize the overhead – and a simple work descriptor shared by threads at the innermost level of parallelism (11, 56). The work in (121) relies on a fixed thread pool, but allows multiple logical threads to be mapped on a single physical thread and maintains a work queue from which threads which become idle can fetch (or steal) work.

The latter approach is based on the widely adopted abstraction of a *work queue* (5, 7): is in fact an orthogonal technique to nesting, and it can be categorized as *tasking*. Once a thread team has been defined, to extract more parallelism it is not necessary to create additional threads: the more lightweight abstraction of the work queue allows existing threads to push and fetch work from there. This offers in many situations a more flexible means to creating parallelism than that offered by nesting alone, thus can be orthogonally adopted to that. The *tasking* (a.k.a. *work-queue*) programming model is well known in the domain of general-purpose computing and in last decade it has been successfully adopted on several multi-core architectures. Cilk (96), Intel Carbon (81), Apple Grand Central Dispatch (7) and OpenMP (106) are successful technologies embodying this model. Recently, some attempts were made to explore its applicability also to heterogeneous systems (i.e., CPU + GPU). The most representative example in this sense is the Fusion series from AMD (5), where a centralized queue system coupled to a task-based programming model enables distributed dispatching of work units between a generic (x86) CPU and a GPU-like accelerator. Programming effort is anyhow significant, since task execution and data transfers must be manually orchestrated using OpenCL. From this point of view, OpenMP tasks (106) are more programmer-friendly, thanks to an annotation-based interface and to the assumption of a uniform memory space (a desirable abstraction also for heterogeneous architectures, pursued by several major vendors).

Currently, there are several freely available open source implementation of the OpenMP 3.x specifications (2), (42), (52). The GCC-OpenMP (GOMP) framework

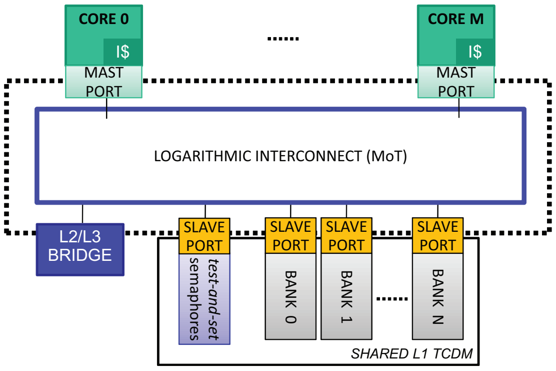


Figure 5.1: On-chip shared memory cluster template

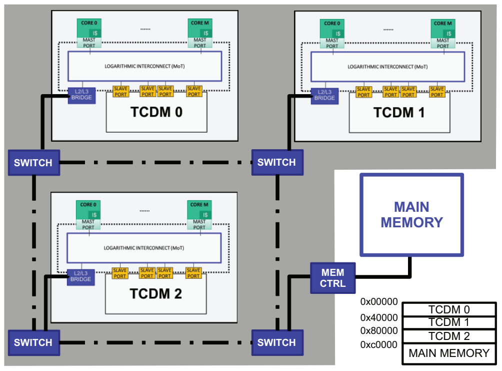


Figure 5.2: Multi-cluster architecture and global address space

(52) implements tasking on top of *pthreads*. The overheads implied by such a layer are significant, as evidenced by many researchers (2, 42).

The cited works target general-purpose computing, using lightweight threading libraries to ensure portability and efficiency. However, embedded platforms are typically more resource-constrained than general-purpose systems, thus requiring different design choices for the implementation of tasking. Indeed, the introduction of an additional threading layer limits the applicability of tasking support to units of work which are coarse enough to pay its overhead. For example, Ayguadé et al.(10) consider tasks with a duration of $10\mu s$ (which considering their 1.67 GHz cores and assuming a CPI of 1 translates in 16K cycles). Similarly, Kumar et al. (81) consider an average of 5K clock cycles for fine-grained tasks. Agathos et al. (2) can afford a 4MB stack for their threads. Clearly, all these numbers need to be significantly scaled down when considering embedded applications and the hardware they run on.

5.3 Shared-memory many-core clusters

Figure 5.1 shows a simplified block diagram of a cluster composed of (up to) 16 RISC-32 processors connected through a low-latency, high bandwidth logarithmic interconnect similar to the ones proposed by Plurality LTD (110) or Rahimi (111). The logarithmic interconnect is built as a parametric, fully combinational Mesh-of-Trees (MoT) (see Figure 5.3).

Processors communicate through a fast multi-banked, multi-ported Tightly-Coupled

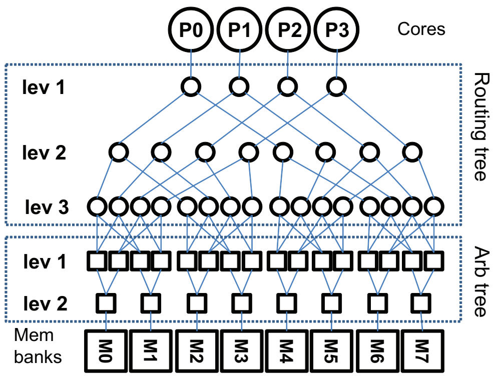


Figure 5.3: Mesh of trees 4x8

Data Memory (TCDM), which is configured as a shared, software-managed scratchpad memory. The number of ports and banks is a multiple of the number of processors to increase bandwidth, by a factor of two or three(25). In case there are no bank conflicts, concurrent accesses by multiple cores to the TCDM are served simultaneously by the MoT. Bank conflicts result in a higher latency, due to contention, which is resolved based on round-robin arbitration. The crossing latency of the MoT is one clock cycle, and word interleaving enables fast concurrent accesses to adjacent memory locations. As a consequence, conflict-free TCDM accesses have two-cycle latency. The interconnection supports read-broadcast: when multiple processors read the same memory location at the same time all the requests are serviced in two cycles.

The L1 scratchpad (TCDM) has limited size of 256KB, thus program code and most of the data are typically stored in larger L2 or L3 memory, while the content of the TCDM is manually updated to the most referenced subset of data at any time. A cluster thus features a L2/L3 bridge for communication with the outer world. This work targets a two-level memory system, with an off-cluster main memory, and we assume a global address space. Scaling to larger system sizes with this architectural template is achieved by interconnecting several clusters through a NoC as shown in Figure 5.2 (see (15)).

Synchronization among the processors is achieved through a segment of the local TCDM address space featuring *test-and-set* semantics. As we will see in Section 5.5, the way the test-and-set memory is physically implemented has a big impact on the performance of the runtime support.

5.4 Multi-level parallelism: nesting

Supporting nested parallelism on a resource-constrained system such as a tightly-coupled clusters is a challenging task. Relying on solutions where new threads are created on the fly whenever more parallelism is needed is not feasible, since this approach would shortly run out of memory, and would impose too large time overheads to enable fine-grained parallelism. Hence, lightweight and highly optimized data structure are necessary. This section provides a detailed analysis of the necessary costs to create additional parallelism at an arbitrary nesting level.

5.4.1 Lightweight Support for Nested Parallelism

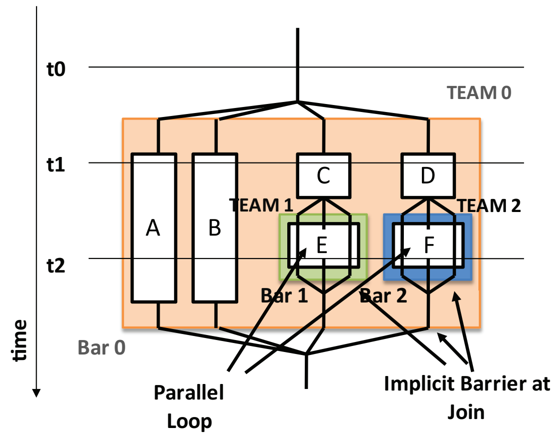


Figure 5.4: Application with nested parallelism

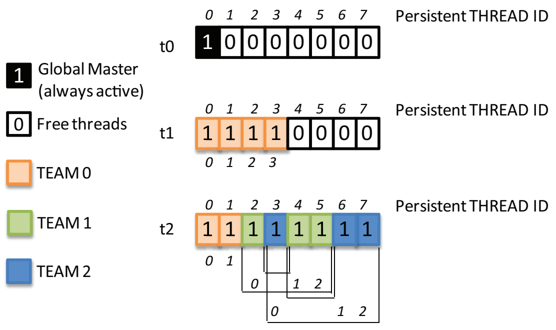


Figure 5.5: Global pool descriptor

The previously FTP (Fixed Thread Pool) approach is the one which provides the simplest requirements for supporting nested parallelism, thus it represents the natural choice for the target architecture. At boot time as many threads as processors are created, providing them with a private stack and a unique ID (matching the hosting processor ID). These threads are called *persistent*, because they will never be destroyed, but will rather be re-assigned to parallel teams as needed. Here it is important to point out that persistent threads are non-preemptive. The thread with the lowest ID is the *global master thread*. This thread will be running all the time, and will thus be

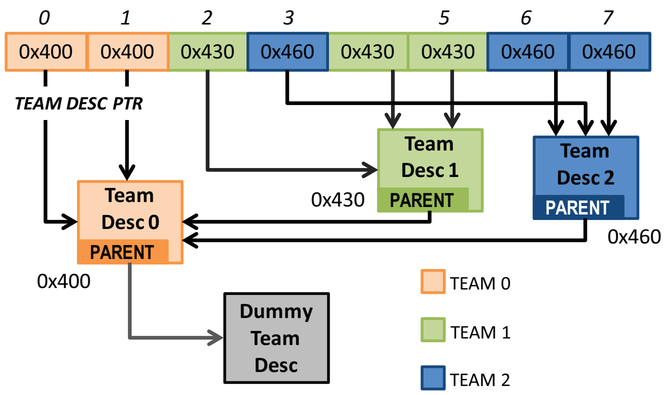


Figure 5.6: Tree of team descriptors to track nesting

in charge of generating the topmost level of parallelism. The rest of the threads are docked on the global pool, waiting for a master thread to provide them with work. At startup, all the persistent threads other than the global master (hereafter called the *global slaves*) execute a microkernel code where they first notify their availability on a private location of a global array (Notify-Flags, or *NFLAGS*), then they wait for work to do on a private flag of another global array (Release-Flags, or *RFLAGS*). The status of global slaves on the thread pool (idle/busy) is annotated in a third global array, the *global pool descriptor*. When a master thread encounters a request for parallelism creation, it fetches threads from the pool and points them to a work descriptor. A detailed description of the various data structures is provided in the following.

5.4.1.1 Forking threads

The first piece of information required by a master to create a parallel team is the status of the global slaves in the pool. As explained, this information in stored in the *global pool descriptor* array. Since several threads may want to concurrently create a new team, accesses to this structure must be locked. Let us consider the example shown in Figure 5.4. Here we show the task graph of an application which uses nested parallelism. At instant *t0* only the global master thread is active, as mirrored by the pool descriptor depicted in Figure 5.5. Then parallel *TEAM 0* is created, where tasks A, B, C and D are assigned to threads 0 to 3. The global pool descriptor is updated accordingly (instant *t1*). After completing execution of tasks C and D, threads 2 and 3 are assigned tasks E and F, which contain parallel loops. Thus threads 2 and 3 become masters of *TEAM 1* and *TEAM 2*. Threads are assigned to the new teams as shown in Figure 5.5 at instant *t2*. Note that the number of slaves actually assigned to a team may be less than what requested by the user, depending on their availability. Besides fetching threads from the global pool, creating a new parallel team involves the creation of a *team descriptor* (see Figure 5.7), which holds information about the work to be executed by the participating threads. This descriptor contains two main blocks:

1. *Thread Information*: A pointer to the code of the parallel function, and its arguments.

2. *Team Information*: when participating in a team, each thread is assigned a team-local ID.

The ID space associated to a team as seen by an application is expressed in the range $0, .., N - 1$, with N being the number of threads composing the team. To quickly remap local thread IDs into the original persistent thread IDs and vice versa, the data structure maintains two arrays. The *LCL_THR_IDS* array is indexed with persistent thread IDs and holds corresponding local thread IDs. The *PST_THR_IDS* is used for services that involve the whole team (e.g., joining threads, updating the status of the pool descriptor), and keeps the dual information: it is indexed with local thread IDs and returns a persistent thread ID. Moreover, to account for region nesting each descriptor holds a pointer to the parent region descriptor. This enables fast context switch at region end.

This team descriptor has a memory footprint of only 48 Bytes. Once the team master has filled all its fields, the descriptor it is made visible to team slaves, by storing its address in a global *TEAM_DESC_PTR* array (one location per thread). Figure 5.6 shows a snapshot of the *TEAM_DESC_PTR* array and the tree of team descriptors at instant *t2* from the previous example.

5.4.1.2 Joining Threads

Joining threads at the end of parallel work typically involves global (barrier) synchronization. Supporting nested parallelism implies the ability of independently synchronizing different thread teams (i.e., processor groups). To this aim we can leverage the mechanism described previously to dock threads, which behaves as a standard *Master-Slave* barrier algorithm (see Section 3.4.2), extended to selectively synchronize only the

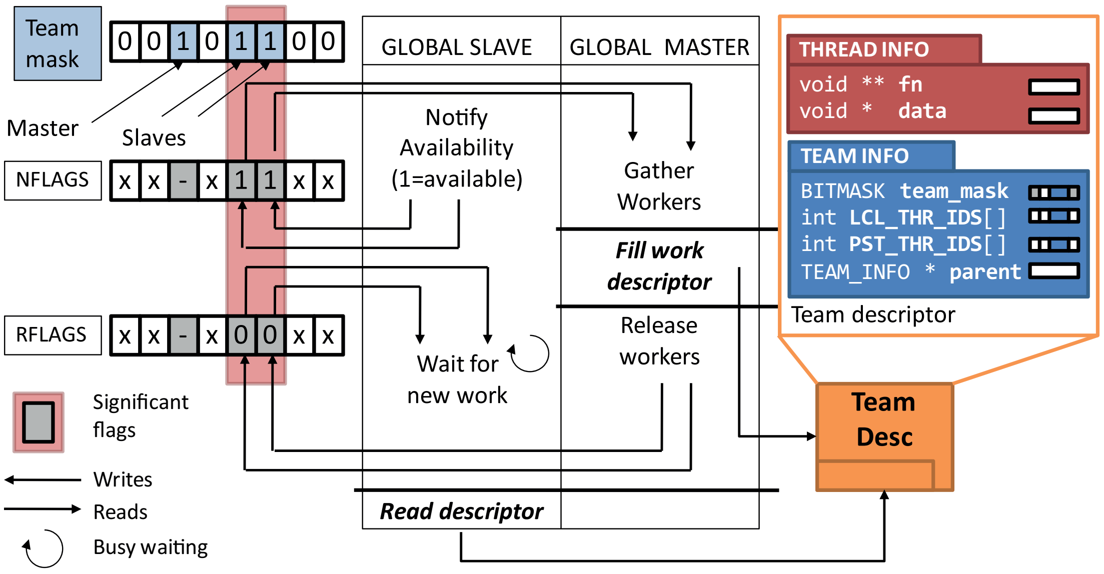


Figure 5.7: Thread docking, synchronization and team descriptor

threads belonging to a particular team. The MS barrier is a two-step algorithm. In the *Gather* phase, the master waits for each slave to notify its arrival on the barrier on a private status flag (*NFLAGS* array). After arrival notification, slaves check for barrier termination on a separate private location (*RFLAGS* array). The termination signal is sent by the master in these private locations during the *Release* phase of the barrier. Figure 5.7 shows how threads belonging to *TEAM 1* (instant *t2* in Figure5.4) synchronize through these data structures.

5.4.2 Experimental validation

The architectural details of target platform are summarized in Table 5.1.

ARM v6 cores	(up to) 16	TCDM banks	16
$I\$_i$ size	1 KB	TCDM size	512 KB
$I\$_i$ line	4 words	L3 latency	50 cycles
t_{hit}	= 1 cycle	L3 size	256 MB

Table 5.1: Architectural parameters

As a first exploration, the cost for opening and closing parallel teams is characterized, providing a breakdown of the various sources of overhead. Two different implementations of thread docking are implemented, namely one which *busy-waits* for available work to do, and one that puts cores to sleep when idling (*idle/wake* in the plots). For the *busy-wait* implementation polling flags for global slaves are allocated on different banks of the TCDM to reduce the conflicts. Figure 5.8 and Figure 5.9 show the cost in (hundred) clock cycles for opening and closing a team, respectively, at the outermost level of parallelism. In this experiment the master thread requests the maximum number of available threads, and we consider increasing sizes for the thread pool. The breakdown plot shows the cost for each of the three main steps taken upon creation of a new team:

1. Allocate and populate the team descriptor.
2. Fetch the slave threads from the global thread pool.
3. Release the slaves from global synchronization structures.

The first component does not depend on the number of threads requested. However, the busy-waiting implementation is subject to the effect of memory bank conflicts. The fact that it is almost insensitive to the polling activity of the slave threads idling on the pool confirms the importance of distributing poll flags on separate memory banks, which eventually make its performance very close to the sleep/wake implementation. On the contrary, the time spent for fetching and releasing slave threads is dependent on their number, since these operations take place from within a loop iterating for as many times as the number of requested slaves.

Overall, it is possible to see that opening a new team composed of 16 threads takes $\approx$ 690 cycles for the busy-wait implementation, and $\approx$ 600 cycles for the sleep/wake implementation.

The breakdown for the team closing shows two components: the time to collect the team threads on the synchronization structure, and the time to tear down the team descriptor and restore the execution context of the parent team by updating global data structures. Collecting threads on the dock is done iterating over the team participants, so the execution time of this section increases with the number of threads in the team. Updating data structures with the information about the parent team context, on the contrary, is independent of the number of threads. It is important to recall here that opening and closing a team implies the use of critical sections to protect updates to

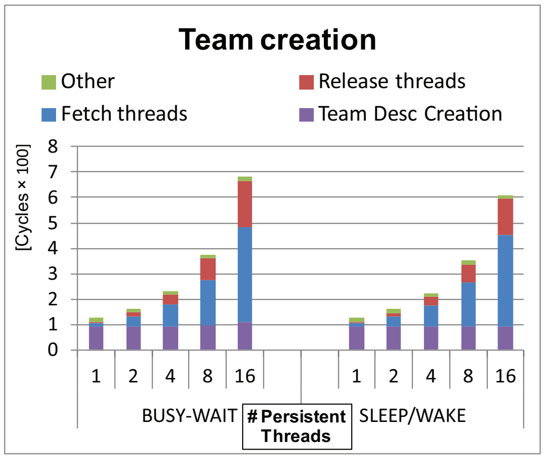


Figure 5.8: Cost of creating a new team

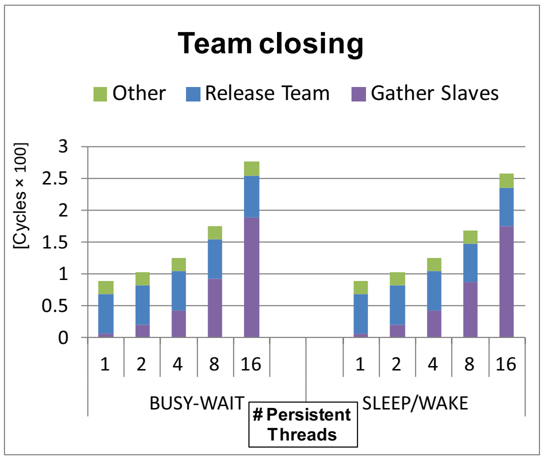


Figure 5.9: Cost of closing a team

global data structures. As such, if more than one attempt to create/destroy a new team at the same time takes place the execution of (parts) of the procedure gets serialized on the concurrent calling threads, which we study next.

Besides characterizing the cost of the basic constructs to create and destroy parallel teams, we will now see the effect of nested parallelism creation from within OpenMP. The library functions invoked by the compiler when a `#pragma omp parallel` construct is encountered have been rewritten as a wrapper around the primitives for parallelism creation. A programming model such as OpenMP exposes a simple and intuitive interface for nested parallelism, however it introduces additional function call overhead to interact with the runtime environment. To measure the OpenMP runtime overhead the EEPC microbenchmarks (134) are used, and their methodology is extended to account for nested parallel regions as described in (40). This methodology basically computes runtime overheads by subtracting the execution time of the parallel microbenchmark from the execution time of its reference sequential implementation. The parallel benchmark is constructed in such a way that it would have the same duration of the reference in absence of overheads.

Figure 5.10 shows the task graph representation of the microbenchmarks used to assess the cost of nested parallelism with depth 1, 2 and 4 respectively. The computational kernel (indicated as W in the plots) is composed uniquely of ALU instructions, to prevent memory effects from altering the measure. We consider a simple pattern where a parallel region is opened, then the block W is executed. This pattern is nested up to 4 times. The thick gray lines in the plots represent the sources of overhead to measure.

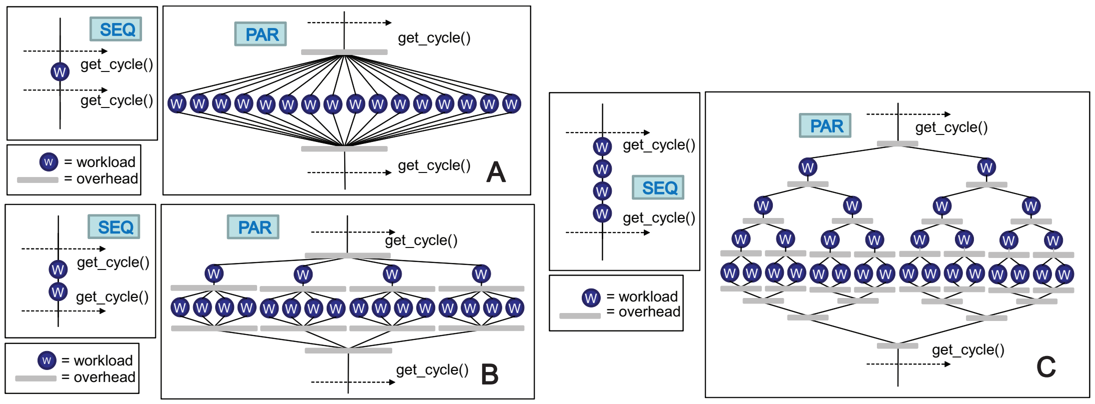


Figure 5.10: Microbenchmark for nested parallelism overhead. A) 1 level, B) 2 levels, C) 4 levels

The difference between the parallel and sequential versions of the microbenchmark represents the total overhead for opening and closing as many parallel regions as the nesting depth indicates. We thus divide the gross overhead by the nesting depth to have an average cost for parallel region opening and close. Figure 5.11 shows this cost for varying granularities of the work unit (W). The first two things to notice are that

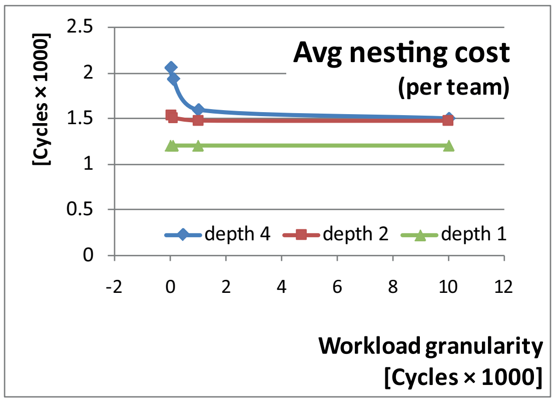


Figure 5.11: Cost for different level of parallelism

i) the cost for single-level parallelism creation from OpenMP is, as expected, slightly higher than the sum of the costs for opening and closing a team that we described earlier, but not much so (roughly 15%), and ii) inner parallelism is slightly costlier to create than the outermost level. The latter is a consequence of the fact that when two or more threads try to concurrently open a new team, the execution of the opening sequence gets serialized due lock-protected updates to global data structures. For this reason, when W contains very small amounts of work this effect is dominant, and the cost for parallelism creation increases with the depth of nesting.

5.4.3 Strassen matrix multiplication

This section evaluates the effectiveness of the nesting support on a real application kernel, and compare it against a single-level parallelization scheme. As outlined in Section 5.4.1, one efficient abstraction that can be orthogonally applied to nesting is the *work queue* abstraction, or *tasking*. OpenMP supports this type of parallelism through dynamic loops (or, in the latest specification, *OpenMP tasks*, that will be extensively analyzed in the following Section). To implement this, the application was rewritten exploiting a single level of parallelism, and workload was partitioned with dynamic loops. We will refer to this scheme as *flat*.

The target application for experiment is the *Strassen* algorithm for matrix multiplication. It is a good candidate for this kind of exploration, since it naturally exposes high degrees of parallelism, both at the task- and data-level, thus being easily parallelized with both the proposed approaches.

The algorithm is shown in the leftmost part of Figure 5.12.

The input matrices A and B are decomposed in four sub-matrices, which can be processed in parallel. Sub-matrices undergo a number of sums/subtractions and multiplications. Each of these operations is fully data-parallel. The algorithm is naturally structured in three stages: in stage one ten sums are computed, which we identify as S_0 ... S_9. These sums can be mapped to parallel tasks, or be data-parallelized. In stage two, seven multiplications are computed (P_1 ... P_7), which similarly exhibit both data and task parallelism. Finally, in the third stage four sets (C_{11} ... C_{22}) of sums and subtractions lead to the final result. The *flat* strategy to parallelize the application with is the following. A single level of parallelism is created using all the threads in the pool. A large parallel region contains all the operations from the three stages in sequence. All of the operations are data parallelized, namely, all the threads can dynamically fetch work from all of the loops. Ideally, this scheme can extract the maximum degree of parallelism, and has a theoretical speedup of $16\times$.

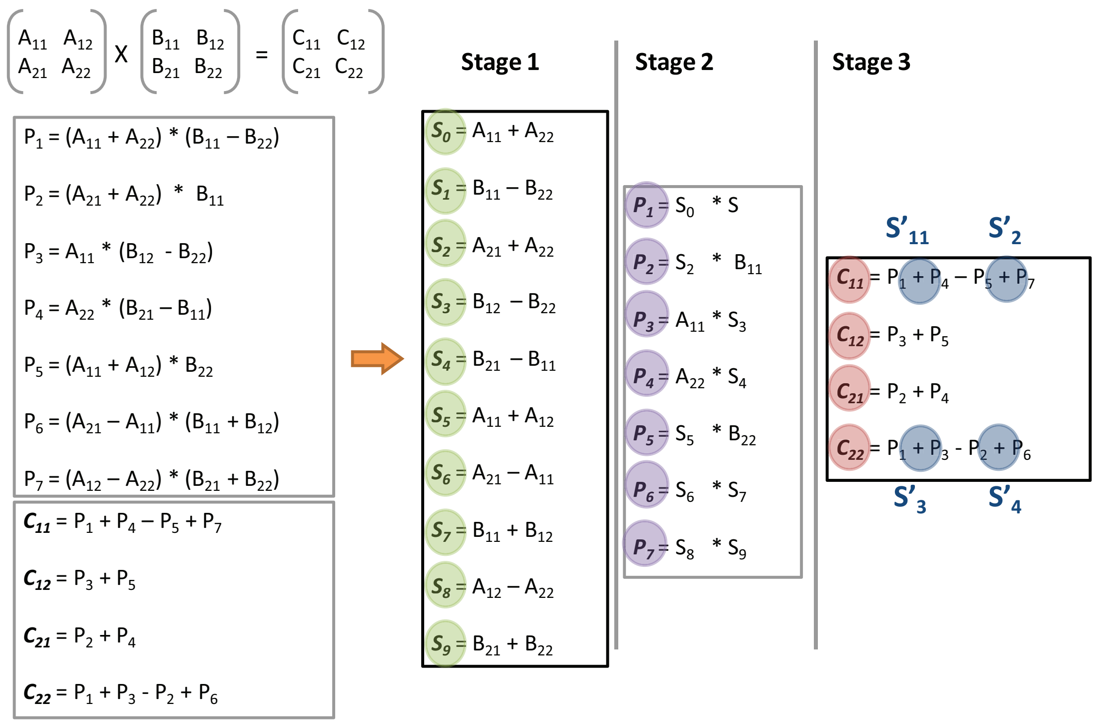
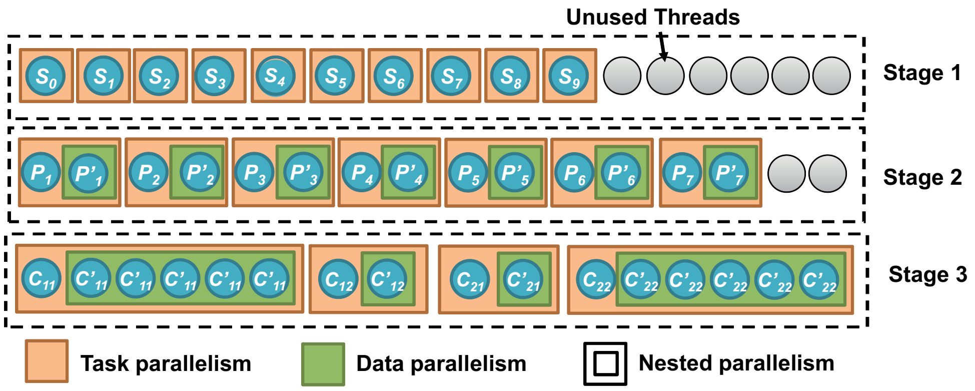


Figure 5.12: Strassen algorithm for matrix multiplication and its basic kernels

Figure 5.13 shows a pictorial representation the *nesting* parallelization scheme, which follows the natural task partition of the application. At stage 1, all the ten

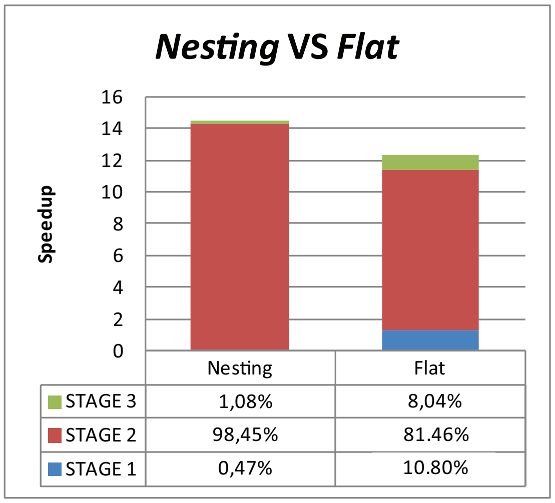


Unused Threads
S0 S1 S2 S3 S4 S5 S6 S7 S8 S9
Stage 1
P1 P'1 P2 P'2 P3 P'3 P4 P'4 P5 P'5 P6 P'6 P7 P'7
Stage 2
C11 C'11 C'11 C'11 C'11 C'11 C12 C'12 C21 C'21 C22 C'22 C'22 C'22 C'22 C'22
Stage 3
Task parallelism
Data parallelism
Nested parallelism

Figure 5.13: Strassen algorithm parallelized with nesting support

sums (S_0, ... , S_9) are assigned to as many threads. No additional data parallelism is created on the remaining six threads, because this would lead to unbalanced execution. These left out threads remain idle in this stage, thus inherently limiting the parallelization speedup to 10×. In the second stage, the seven multiplications ($P_1, \ldots, P_7$) are initially assigned to seven threads. Each of these threads generates a nested region and exploits an additional thread thus leveraging data parallelism as well. The remaining two threads in the global pool are left idle, thus the maximum achievable speedup is limited to 14x. In the third stage, four parallel threads are assigned the final sums ($C_{11}, \ldots, C_{22}$). The workload contained in these tasks is unbalanced by a factor of 3:1 for tasks C_{11} and C_{22} with respect to the other two (three sums instead of one). By creating nested data-parallel regions with different number of threads (C_{11} and C_{22} will run on six threads, while C_{12} and C_{21} will run on two) the runtime is capable of balancing the workload and exploiting all available threads. This stage fully exploits the computational resources of the system, with a theoretical speedup of 16x.

Both the *nesting* and *flat* parallelization schemes were implemented in two variants, with coarse-grained and fine-grained tasks, by sizing accordingly the chunk of loop iterations. As a first experiment, let's consider an instance of the algorithm using 64x64



	Nesting	Flat
🟩 STAGE 3	1,08%	8,04%
🟥 STAGE 2	98,45%	81.46%
🟦 STAGE 1	0,47%	10.80%

Figure 5.14: *Nesting* and *flat* speedup

matrices (four 32x32 submatrices). Results for this experiment are shown in Figure 5.14, where the speedup achieved by the two parallelization strategies is reported. Overall, the theoretical speedup for the *nesting* approach is ≈14× (16× for *flat*). It is possible to see that, notwithstanding the threads left idle at times, the *nesting* approach matches its theoretical speedup. The *flat* approach, on the contrary, is far from it. The numbers on the table below the figure show the percentages of time spent in the three stages. It is possible to notice two things. First, the multiplication kernels unsurprisingly dominate execution time. Second, the first and third stages take non negligible time with the *flat* approach as compared to *nesting*. This is attributable to the overhead for distributing workload from the work-queue at a too fine granularity.

In the following we "zoom-in" these phases to have better insight. Leftmost plot in Figure 5.15 shows how the execution time of the first stage is affected by the size of the input submatrices, which are set to 32x32, 64x64 and 128x128. This plot confirms that for fine-grained workload (leftmost plot) the *flat* approach cannot achieve any speedups. To see how this phenomenon can be mitigated by considering coarser work units we increased the chunk size for the parallel loop to its maximum (the number of iterations is evenly divided among participating threads). Even in this case (plot in the middle), if the matrix size is too small the overhead for the work queue is not amortized. With bigger matrix sizes (128x128) we achieved a 5× speedup. All of those results are far from the theoretical speedup achievable with the *loop* parallelism because of the implementation overheads. The rightmost plot shows how the *nesting* approach achieves much better results. For matrix sizes of 128x128 this parallelization scheme achieves its theoretical speedup peak.

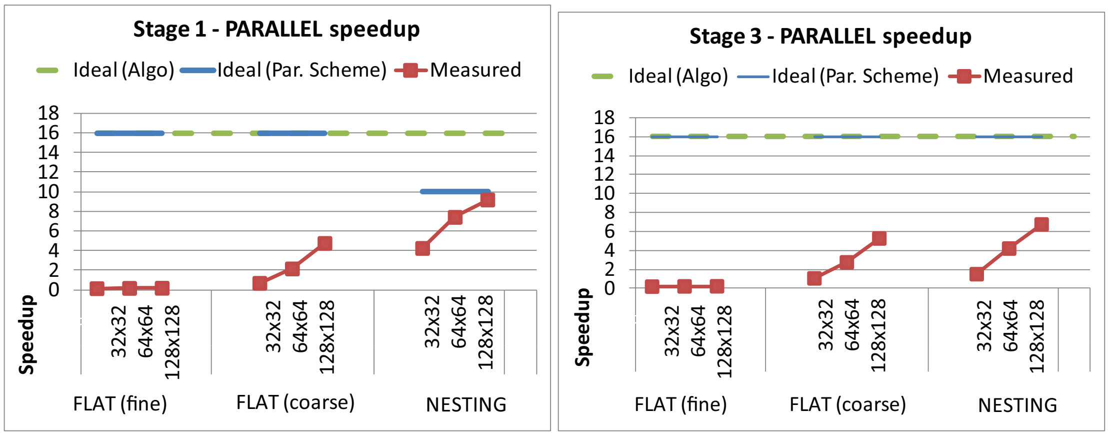


Figure 5.15: Effect of task granularity on speedup for Strassen application

Similar plots are provided for the third stage.

5.5 Irregular and dynamic parallelism: tasking

OpenMP constructs for dynamic parallelism provide a powerful and flexible solution to exploit irregular parallelism in target applications, but their practical implementation requires sophisticated runtime system support, which typically implies important space and time overheads. The applicability of the approach is thus often limited to applications exhibiting units of work which are coarse-grained enough to amortize these overheads. This section describes the design of an optimized runtime environment supporting fine-grained tasks on an embedded shared-memory cluster. An extended analysis of the semantics of the programming model is performed, with the aim of identifying key performance bottlenecks and finding solutions for solving them, and *ad-hoc* architectural (HW) extensions are proposed for it.

5.5.1 Analysis of OpenMP Tasking

OpenMP 3.0 introduces a task-centric model of execution. The new `task` construct can be used to dynamically generate units of parallel work that can be executed by every thread in a parallel team. When a thread encounters the `task` construct, it prepares a task *descriptor* consisting of the code to be executed, plus a data environment inherited from the enclosing structured block. `shared` data items point to the variables with the same name in the enclosing region. New storage is created for `private` and `firstprivate` data items, and the latter are initialized with the value of the original variables at the moment of task creation. The execution of the task can be immediate or deferred until later by inserting the descriptor in a *work queue* from which any thread in the team can extract it. This decision can be taken at runtime depending on resource availability and/or on the scheduling policy implemented (e.g., breadth-first, work-first (42)). However, a programmer can enforce a particular task to be immediately executed by using the `if` clause. When the conditional expression evaluates to *false* the encountering thread suspends the current task region and switches to the new task. On termination it resumes the previous task. Specifications also enable work-unit based synchronization. The `taskwait` directive forces the current thread to wait for the completion of every tasks generated from the current task region. *Task scheduling points* (TSP) specify places in a program where the encountering thread may suspend execution of the current task and start execution of a new task or resume a previously suspended task.

Figure 5.16 shows a layered approach to designing the primitives for the tasking constructs. These constructs are depicted in the top layer blocks (in red). OpenMP

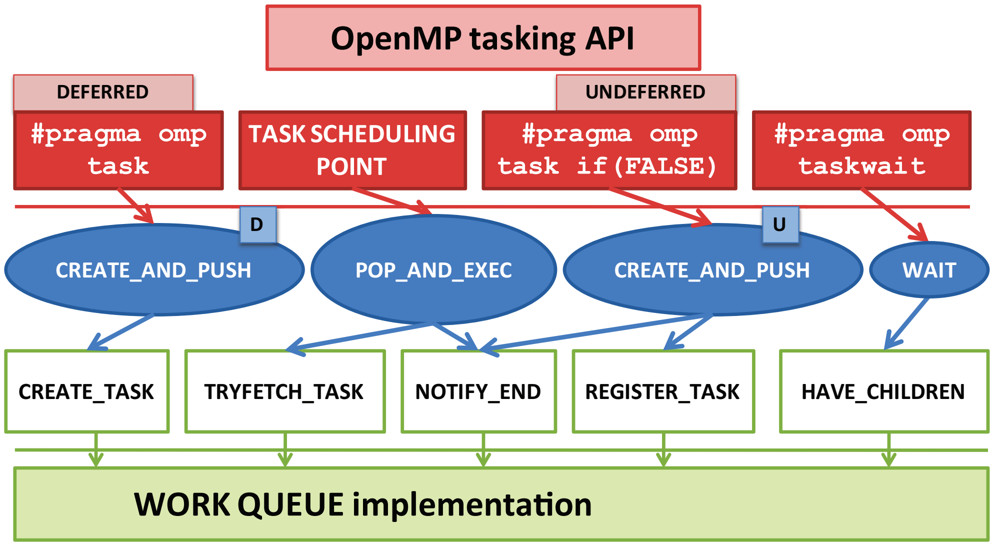


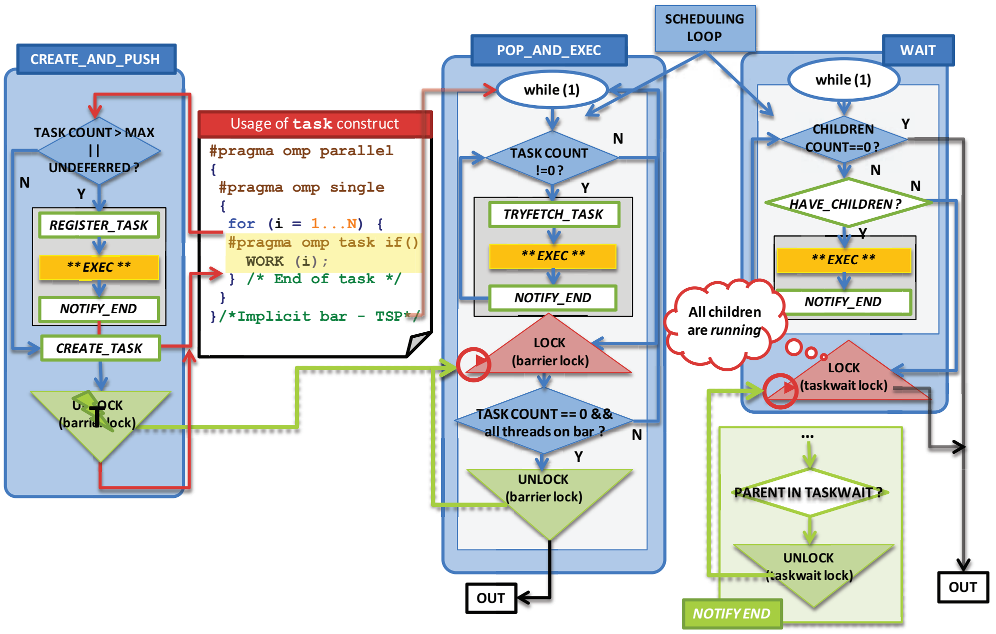
Figure 5.16: Design of tasking support

tasks are managed by a main *work queue* where units of work can be pushed to and popped from (bottom layer block). The gap between OpenMP directives and the *work queue* is bridged by an intermediate runtime layer (blue blocks), which operates on the queue through a set of basic primitives (white blocks) to implement the semantics of the tasking constructs.

The proposed runtime design relies on a centralized queue with breadth-first, LIFO scheduling. Tasks are tracked through *descriptors* which identify their associated *task regions* and which are stored in the *work queue*. The two basic operations on the queue are task insertion and extraction. Inserting a task has two effects: i) creating a new descriptor for it, and ii) registering it as a child of the executing task (its *parent*). These semantics are formalized as a primitive called `CREATE_TASK`.

Extracting a task from the *work queue* retrieves its descriptor for execution. This is formalized with a `TRYFETCH_TASK` primitive, which returns the task descriptor in case of successful extraction, or a NULL pointer if the *work queue* is empty. Task extraction should only return the descriptor to the caller, not detach it from the *work queue* until

the task has completed execution. This is necessary for correctly supporting synchronization (`taskwait`). Thus, a separate NOTIFY_END primitive is envisioned to dispose of the descriptor, which acts as an epilogue to task execution.
Note that since the TRYFETCH_TASK primitive does not remove the task *descriptor* from the *work queue*, it is necessary to mark it as *running* to avoid multiple extractions of the same *descriptor*. Thus, the CREATE_TASK inserts a *waiting* task in the *work queue* and the TRYFETCH_TASK changes its status to *running*. NOTIFY_END marks it as *ended*.
To support undeferred tasks (e.g., whose `if` condition is evaluated to *false*), a REGISTER_TASK primitive is introduced, which inserts a *descriptor* marked as *running*.
Finally, the HAVE_CHILDREN primitive allows to determine if a task has children not yet assigned to a thread (i.e., in the *waiting* state). As we will see in the next section, this is necessary to implement task switching capability in presence of a `taskwait`.



```
#pragma omp parallel
{
 #pragma omp single
 {
  for (i = 1...N) {
  #pragma omp task if()
    WORK (i);
  } /* End of task */
 }
}/*Implicit bar - TSP*/
```

Figure 5.17: Design of task scheduling loop

5.5.2 Design of the runtime layer

Let us consider the simple example of the `task` construct in the code snippet of Figure 5.17. The `parallel` directive creates a team of worker threads, then only one of them executes the `single` block. This thread acts as a work producer, since it is the only one encountering the `task` construct. The control flow for the rest of the threads falls through the parallel region to the implied barrier at its end.

The most important part of the implementation of the tasking execution model is Task Scheduling Points (TSP). Parallel threads are allowed to switch from one task to another:

1. at `task` constructs;
2. at implicit and explicit barriers;
3. at the end of the current task;
4. at `taskwait` constructs;

The first point prevents system oversubscription in cases where a thread is required to generate a very high number of tasks (e.g., the `task` directive is nested inside a loop with a huge number of iterations). Placing a TSP on a `task` construct allows the producer thread to switch to executing some of the tasks already in the queue. Task creation is resumed once the queue has been depleted to a certain level.

To keep the implementation of task scheduling as simple as possible, upon encountering a `task` directive, threads calls the `CREATE_AND_PUSH` runtime function, depicted on the left part of Figure 5.17. Here, the caller first checks for the number of tasks already in the queue. If this number exceeds a given threshold the thread does not insert the task in the queue, but it immediately executes it instead. Note that this can not be implemented through a simple jump to the task block code. Executing a task without creating a descriptor and connecting it to the others will in fact result in ignoring its existence, which may lead to incorrect functioning of the `taskwait` directive due to bad internal representation of the task hierarchy. Thus the act of creating and inserting in the queue a descriptor for a *running* task is formalized by the `REGISTER_TASK` primitive. Similarly, task execution termination is signaled through a call to `NOTIFY_END`.

This same solution is adopted when an undeferred task is explicitly generated by the user through the `if(FALSE)` clause. In all the other cases, a call to `CREATE_AND_PUSH` will result in regular creation of a team descriptor and insertion in the queue (`CREATE_TASK`). After that, the producer thread signals the presence of work in the queue by releasing a *barrier_lock* on which consumer threads wait.

This brings us to the second TSP. As explained before, threads not executing the `single` block are trapped on the barrier implied at the end of the region. This is implemented through a call to the `POP_AND_EXEC` function (central part of Figure 5.17). Here, threads first check for the presence of tasks in the queue. If there are tasks available the encountering thread initiates an execution sequence. First, the task descriptor is extracted from the queue with the `TRYFETCH_TASK` primitive. Then, the associated task code is executed. Finally, notification of task completion is signaled through the `NOTIFY_END` primitive. If the queue is empty, the encountering thread busy waits on the *barrier_lock* (note that this lock is initialized as *busy* at system startup). When the lock is released by a producer pushing a task in the queue, the current thread checks for the presence of tasks in the queue and for the number of threads waiting on the lock (annotated in a counter). If all threads are on the lock and there are no tasks in the queue, this indicates that the end of the parallel region has been reached. Otherwise, there may still be work left to do, so the thread jumps back to the scheduling loop.

Note that upon task termination, an iteration of the scheduling loop is again performed, thus implementing the third TSP.

Finally, a TSP is also implied at a `taskwait` construct. However, in this specific case the *Task Scheduling Constraint* only allows to switch execution to a task that was directly created by the current one to prevent deadlocks. This semantics are implemented in the `WAIT` runtime function. Each task keeps track of its children. The `HAVE_CHILDREN` primitive allows to fetch the descriptor of a child task in the *waiting* state. If a valid task descriptor is returned, the thread can be rescheduled on that task. Otherwise, all the children are in the *running* state and the thread will have to stay idle waiting for their completion. In this case, the last terminating child notifies the parent through the `NOTIFY_END` primitive.

5.5.3 Implementation details

Task descriptors are interconnected within two co-existing data structures; a *queue* and a *tree*. The *queue* contains the descriptors of all the tasks in the *waiting* state for the current parallel region and it is implemented as a doubly-linked list. Similarly, to build the *tree* representation, each descriptor handles a reference to a doubly-linked list of children, i.e., the set of tasks that it has previously created, being either in the *waiting* or *running* state. Each descriptor also traces the *parent* task. Upon task creation, the corresponding descriptor is inserted into the *work queue* by updating the connections in the *queue* and in the *tree* data structures.

Consistency between the two representations is enforced by making their updates atomic through a *work_queue_lock*. Each of the insertion/removal primitives protects the critical sections that update the descriptor with this lock. The assembly was manually optimized for the primitives to minimize the duration of critical sections.

To ensure fast access to task descriptors, they are stored in L1 memory. However, the number of tasks co-existing in the system can become very high, thus a mechanism is needed, which avoids memory oversubscription. The solution is to use a custom allocator with a fixed number (1024) of statically reserved bins in a region of the TCDM. The `task_malloc` and `task_free` retrieve and dispose memory for a new descriptor using a LIFO list of free bins. Concurrent write accesses to the list are protected by a lock (*task_malloc_lock*).
There are four main types of locks in the tasking support framework. Besides *task_malloc_lock* and *work_queue_lock*, the *barrier_lock* is used inside the task scheduling loop for idle threads to wait for available tasks and the *taskwait_lock* is used by a task to wait for termination of its children.

Note that there is one *work_queue_lock* and one *barrier_lock* for each parallel region in the system, while there is a *taskwait_lock* for every task in the system. In fact, the number of locks used at any time can be high. As a consequence, the single-ported test-and-set (TAS) memory may easily become a bottleneck if multiple threads are concurrently performing any form of synchronization.

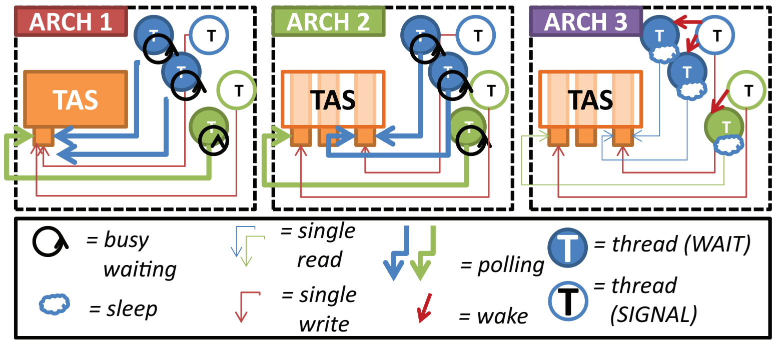


Figure 5.18: Different architectural configurations (*ARCH*) of the TAS memory

To address this issue the following architectural variants are considered. They are shown in Figure 5.18. The implementation with single-ported, single-banked TAS memory is referred to as **ARCH 1** and it is considered as a baseline for the other solutions. Any *wait* operation in this architecture is always implemented with busy-waiting (see leftmost part of the figure). As the number of locks increases, the concurrent traffic overloads the TAS port. However, in many cases the conflict is created by contention for the memory port, not for a lock. This issue can be mitigated by considering an architectural modification to increment the number of ports and banks of the TAS memory. In this variant (referred to as **ARCH 2**) the TAS segment has 16 banks/-ports and thus, similar to the data TCDM segment, can serve concurrent accesses to different locks in parallel.

In both **ARCH 1** and **ARCH 2** all of the *wait* operations are implemented with busy-waiting on the lock until the corresponding *signal* operation FREEs it. While **ARCH 2** solves the issue of sequentialization of accesses to distinct locks, it does not remove the polling activity of multiple cores, which creates congestion. While *work_queue_lock* and *task_malloc_lock* are used to implement critical sections protecting atomic queue updates, *barrier_lock* and *taskwait_lock* implement a different synchronization pattern, where one thread (or more) is waiting for another one (or more) to notify verification of a specific event. Thus, while in the first case a *busy-waiting* implementation is to be preferred (short duration of the critical section), in the second case it could rather be beneficial an alternative *idle/wake* mechanism where threads that find a busy lock enter a sleep state and will be awaken after the lock has been set to FREE. We will refer to this architectural variant as **ARCH 3**.

5.5.4 Experiments

To validate the runtime design, an extensive set of experiments was performed using a SystemC-based virtual platform modeling the tightly-coupled cluster described in Section 5.3 (26). Table 5.2 summarizes the main architectural parameters, a typical setup for the considered platform template (see (15), (25)). In this section three types of experiments are shown:

1. cost characterization of the main tasking constructs;
2. parallelization speedup for varying task granularity and comparison with other tasking implementations;

Table 5.2: Architectural parameters

ARM v6 cores	16	TCDM banks	16
I$ size	1 KB	TCDM latency	$\geq$ 2 cycles
I$ line	4 words	TCDM size	256 KB
t_{hit}	= 1 cycle	L3 latency	$\geq$ 60 cycles
t_{miss}	$\geq$ 59 cycles	L3 size	256 MB

3. parallelization speedup for two real programs: the Strassen matrix multiplication benchmark and the FAST corner detection application.

5.5.4.1 Tasking cost characterization

Here, the cost of the OpenMP tasking services is measured. 16 threads were created, one per processor, with one of them producing 256 tasks. The tasks are composed of ALU instructions only, to exclude memory effects from the measurement (each task consists of 500 ALU operations).

Figure 5.19 shows the results for these measurements for each of the three architectural variants discussed in the previous section. There is one additional bar per plot, labeled IDEAL, which shows the cost for executing the corresponding runtime operation on a single core, while the rest of the cores is idling (thus no interference of any kind takes place). These experiments are run under architectural variant **ARCH 1**.

A first observation is that the cost for tasking in the IDEAL case is between 70 and 130 clock cycles. The optimized assembly routines allow a low cost for these services.

Figure 5.19 (a) shows the cost for creating a task with the `task` directive. Contributions include time for creating the descriptor (*task_malloc*) and initializing it (*desc init*), *work_queue_lock* acquisition (*lock*) and release (*unlock*), plus update of the internal work queue data structures (*update wq*). Results for **ARCH 1** show that the duration of the *lock* and the *unlock* phases are greatly impacted by high contention for the single-ported TAS memory, as well as task descriptor creation and initialization (in which some locks are accessed). When moving to **ARCH 2** most of this effect disappears as expected. **ARCH 3** further improves the results, since the polling traffic for threads waiting on the *barrier_lock* is removed, thus reaching the IDEAL performance.

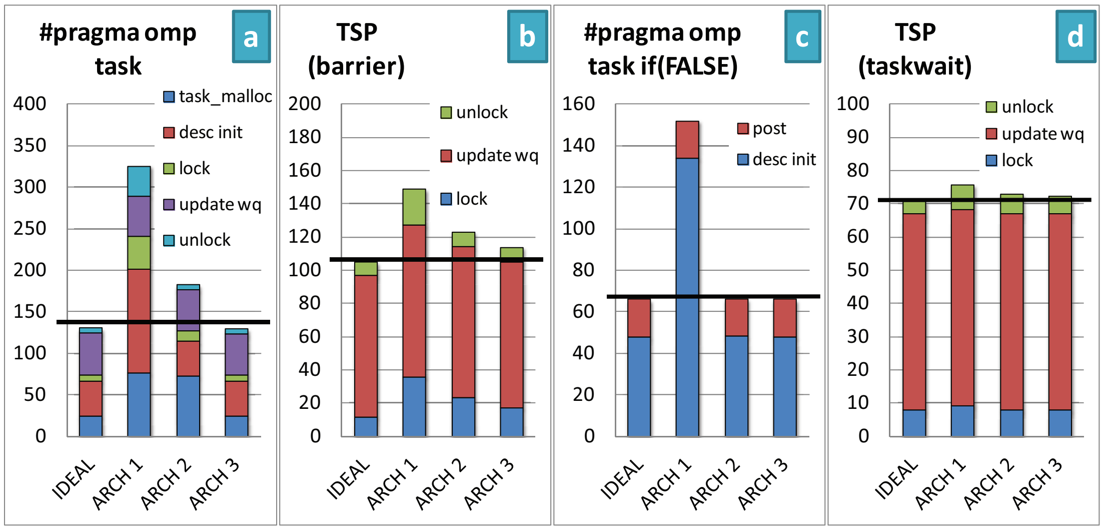


Figure 5.19: Breakdown of the time spent in high-level OpenMP services

Figures 5.19 (b), (c) and (d) report similar cost results respectively for i) a Task Scheduling Point occurring on implicit and explicit *barriers* ii) creating an undeferred task (annotated with a `if(FALSE)` clause) iii) a Task Scheduling Point occurring on a `taskwait`. Similar conclusions hold for the benefits of **ARCH 2** and **ARCH 3** over **ARCH 1**. Note that for undeferred tasks there is no need to acquire a lock since the descriptor is in a local variable, which also removes the need for `task_malloc/free`. The *post* cost refers to switching execution back to the calling context.

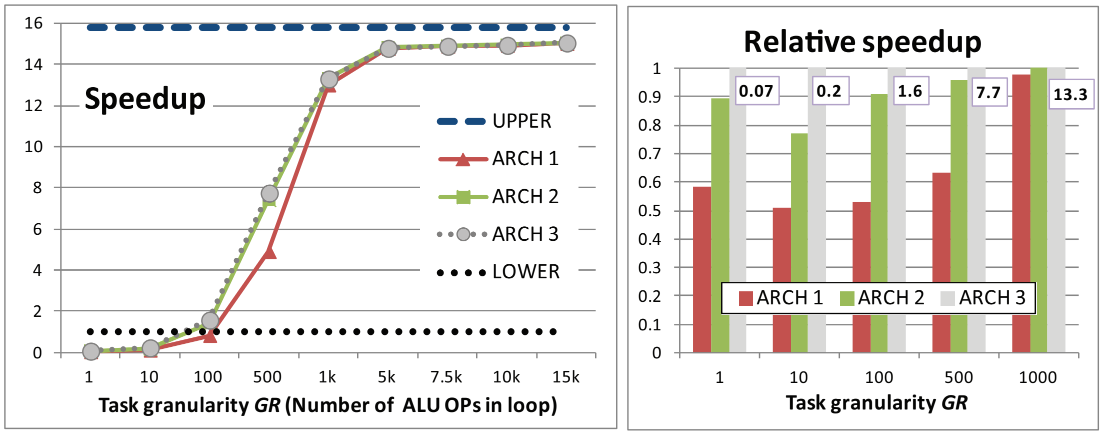


Figure 5.20: Parallelization speedup for increasing task granularity

5.5.4.2 Task granularity impact on speedup

Figure 5.20(a) shows how different task granularities affect speedup for each of the three architectures. For this characterization a synthetic benchmark is considered, which consists of a loop with a parameterizable number of iterations (GR) and whose body contains one dummy ALU (MOV) instruction. The setup is the same of the previous experiment, with 16 threads, where only one is responsible for the creation of 256 tasks while the remaining 15 can immediately start to execute them (the producer thread can also join task execution after creating them all). Experiments were performed for task granularities (GR) varying in the range between 1 and 15K. To obtain the speedup the reference is the total execution time for the 256 tasks on a single thread with the parallel execution time. The theoretical maximum speedup (16$\times$) is depicted by the **UPPER** curve, while the **LOWER** curve shows a lower bound to the speedup (i.e., below this value there is slowdown).

The figure shows that the **LOWER** value is reached for values of GR $\approx$80, while the **UPPER** bound is asymptotically reached for granularities of $\approx$5000. Note that the actual maximum speedup is lower than 16$\times$, because one processor acts as a task producer and does not take part to the actual parallel execution. This limits the maximum speedup achieved in slightly more than 15$\times$. In this region there is no significant difference among the architectures. For finer tasks, however, the performance of different architectures differentiate. Figure 5.20 (b) “zooms in” the finer task region, plotting the relative speedups referred to the **ARCH 3** for a given granularity. The numbers on top report the absolute values for the speedup of **ARCH 3**. A considerable speedup (20 to 30%) is achieved when switching from **ARCH 1** to **ARCH 2**, at any granularity. Switching to **ARCH 3** further improves performance, up to +30% speedup (for GR$\approx$ 10).

In summary, these experiments identify 80 ALU instructions as the minimum task granularity to achieve any speedup in the proposed implementation and 5000 to reach the upper bound. Note that tasks in real applications are likely to have more than 80 instructions, including memory accesses, here not modeled.

The proposed tasking support was also compared to GCC-OpenMP (52) and OMPi (2). Experiments were performed on a Intel i7 quad-core machine @3.4GHz, featuring HyperThreading technology. Since the target machine only has four cores, for fair

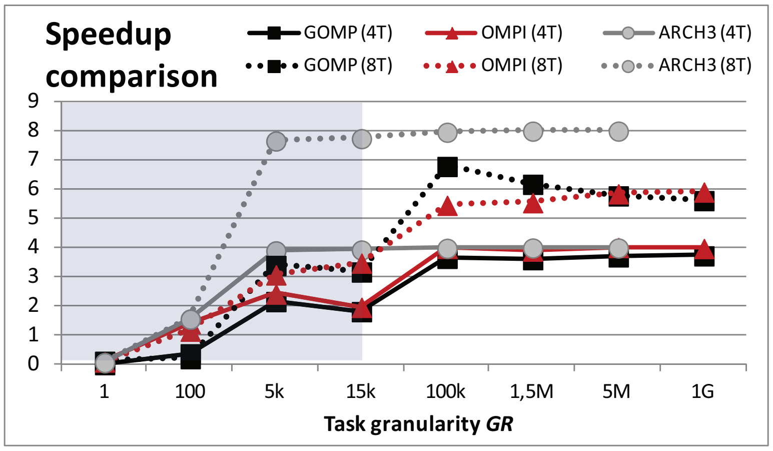


Figure 5.21: Parallelization speedup against existing OpenMP runtimes (c)

comparison the experiment was repeated on an instance of the target cluster with the same number of PUs. In addition, HyperThreading (8 threads) was also compared against with the runtime running on 8 PUs. The results for this experiment are shown in Figure 5.21, where the gray area on the left matches the one of in Figure 5.21. Solid lines refer to the experiment with four cores, dashed lines refer to the experiment with 8 cores. The results show that **ARCH 3** asymptotically reaches the maximum speedup (4×) for GR≈5000, outperforming both GOMP and OMPi which reach their peak values for GR≈100000. Similar conclusion applies when 8 threads are considered. Note that, since the synthetic tasks are made of ALU instructions, HyperThreading only achieves 6×. Results prove that the proposed solution achieve peak speedup for tasks ≈ 20× finer-grained than both GOMP and OMPi.

5.5.4.3 Real Benchmarks

In this section the runtime is validated on two real programs: *Strassen* matrix multiplication and FAST corner detection.

The Strassen algorithm was already partitioned following the *fine* scheme shown in Section 5.4.3, with the exception that OpenMP tasks were used instead of dynamic loops. Figure 5.22 shows how the speedups for the two tasking strategies scale in the different architectures, as the size of matrices increases and for each stage. Ideal speedups are also reported in dotted black lines. Obviously the speedup increases with matrix size, since the overhead of tasking becomes less significant as the the actual

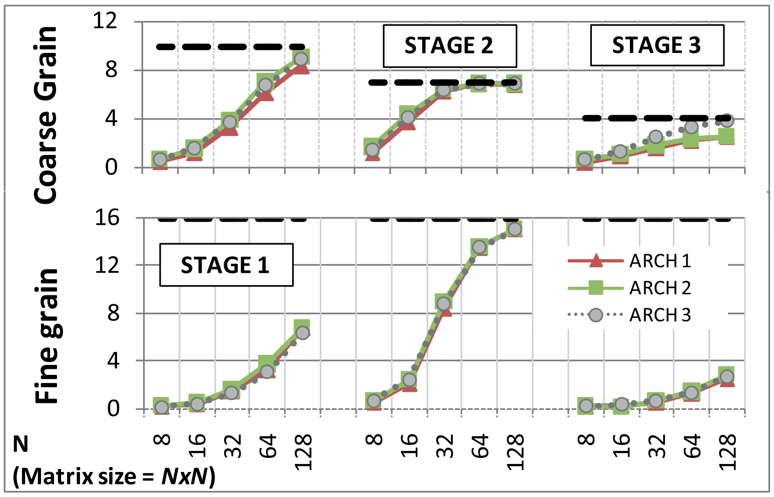


Figure 5.22: Speedup of Strassen Algorithm

workload grows. This is the reason why in Stage 2 (more computation) the runtime reaches near-ideal speedup for small matrices (32x32 for the COARSE scheme, 64x64 for the FINE scheme). However, the FINE strategy does not always allocate enough work to amortize the overheads, and this is the reason why Stages 1 and 3 do not scale beyond 7× and 4×, respectively.

As shown in the charts, considering different architectures does not significantly affect performance for Stages 1 and 2. This is due to the regular nature of the parallel workload, which does not require synchronization. Stage 3, on the contrary, uses a `taskwait` construct to separate the two sub-stages, thus showing the benefits of **ARCH 3**.

The FAST (112) algorithm compares the intensity value of each point p of the input image with all the sixteen points on the circle of radius 3 and center p. p is classified as a corner if there exists a set of contiguous pixels within the circle that are all brighter (minimum) or darker (maximum) than p (with a tolerance threshold). The number of contiguous pixels and the threshold value are both algorithm parameters; typical values are respectively 9 and 20.

Given an $N*M$ input image, the algorithm generates an output vector whose size is $N*M*3$, containing the coordinates of the corner points and a score. The latter is used in a subsequent non-maxima suppression stage, which merges multiple pixels belonging to the same corner. Finally, a keypoint detection pass detects relevant features.

The core kernel performs most of the computation and it exhibit data-parallelism at the pixel level. By estimating the number of instructions in the main loop body, it is easy to determine a minimum number of iterations to achieve near-ideal speedup by checking the chart in Figure 5.20. To achieve this goal, the tasks were designed to process an entire image row. Experiments were performed increasing the size $N \times N$ of input images, with $N \in \{64, 128, 256, 512\}$, thus the granularity of tasks doubles with the input size. However, due to the limited size of the TCDM it is not possible to store the whole dataset therein. The image is split into *stripes*, which are processed one after the other. The double buffering technique overlaps computation and DMA transfers from the global memory. Table 5.3 shows the speedup of the parallelized algorithm compared to the sequential version for different image sizes. A considerable speedup is achieved even for small images (11× for a 64x64 image, with each task only processing 64 pixels) and the speedup reaches 91% of the theoretical 16× for $N \geq 256$.

Table 5.3: Speedup of the parallel FAST algorithm

Input dim	64x64	128x128	256x256	512x512	Ideal
Speedup	11,01	13,54	14,19	14,60	16

5.6 Hardware support for irregular parallelism

OpenMP constructs for dynamic parallelism provide a powerful and flexible solution to exploit irregular parallelism in target applications, but their practical implementation requires sophisticated runtime system support, which typically implies important space and time overheads. The applicability of the approach is thus often limited to applications exhibiting units of work which are coarse-grained enough to amortize these overheads. In this section, the major sources of overhead in the implementation of OpenMP *dynamic loops*, *sections* and *tasks* are studied, and hardware implementation is proposed for a generic Scheduling Engine (HWSE) which fits the semantics of the three constructs. The adaptability of this HW block in the context of different programming models is also discussed. The HWSE is designed as a tightly-coupled block to the PEs within a multi-core cluster, communicating through a shared-memory interface. This allows very fast programming and synchronization with the controlling

PEs, fundamental to achieving fast dynamic scheduling, and ultimately to enable fine-grained parallelism. The HWSE was modeled in RTL, to obtain accurate synthesis results, and in SystemC, to validate the proposed approach by running run complete applications. Results are compared against two runtime implementations, OMPi (2) and GNU LIBGOMP (52).

5.6.1 Analysis of OpenMP dynamic parallelism support

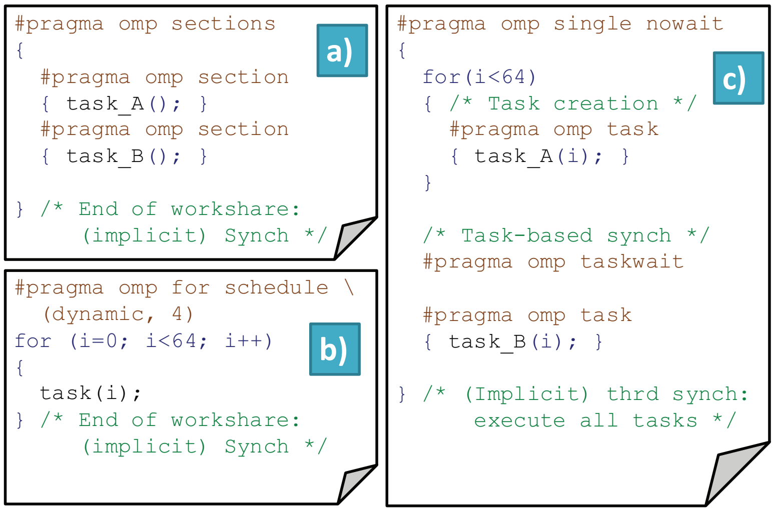
Here, the OpenMP constructs for dynamic parallelism are analyzed, aiming at i) deriving a minimal set of primitives that capture their semantics and ii) characterizing the cost of each primitive, to discover best candidates for hardware acceleration. The reference OpenMP implementation considered in this work is the GNU GCC runtime library (`libgomp` (52)). This will be used as a baseline also in the evaluation section.

OpenMP features three different constructs to support dynamic parallelism: *sections*, *dynamic loops* and *tasking*.

1. **Sections** (Figure 5.23 (a)). Different portions of code are annotated to statically decompose a program into coarse-grained tasks (here, `task_A` and `task_B`) deployed onto parallel threads;
2. **Dynamic loops** (Figure 5.23 (b)). Tasks are dynamically created out of chunks of loop iterations and executed by parallel threads.
3. **Tasking** (Figure 5.23 (c)). OpenMP *Tasks* have been introduced since specifications 3.0 (106). Compared to *sections*, OpenMP *tasks* enable more sophisticated forms of dynamic, irregular and asynchronous parallelism.

Chapter 2 explains in details each construct. Their basic usage is shown in Figure 5.23.

Figure 5.24 shows how the code in Figure 5.23 is transformed by the GCC compiler. The compiler resorts to the runtime system to retrieve the next available chunk of loop iterations for dynamic loops (`GOMP_loop_dynamic_next()`) or the next available section (`GOMP_section_next()`). Both functions implement a FIFO queue, to which parallel threads access in a mutually exclusive manner to update a shared counter. *sections* and *dynamic loops* rely on a *work-share* data structure, which describes the parallel



```
#pragma omp sections
{
  #pragma omp section
  { task_A(); }
  #pragma omp section
  { task_B(); }

} /* End of workshare:
     (implicit) Synch */
```

```
#pragma omp for schedule \
  (dynamic, 4)
for (i=0; i<64; i++)
{
  task(i);
} /* End of workshare:
     (implicit) Synch */
```

```
#pragma omp single nowait
{
  for(i<64)
  { /* Task creation */
    #pragma omp task
    { task_A(i); }
  }

  /* Task-based synch */
  #pragma omp taskwait

  #pragma omp task
  { task_B(i); }

} /* (Implicit) thrd synch:
     execute all tasks */
```

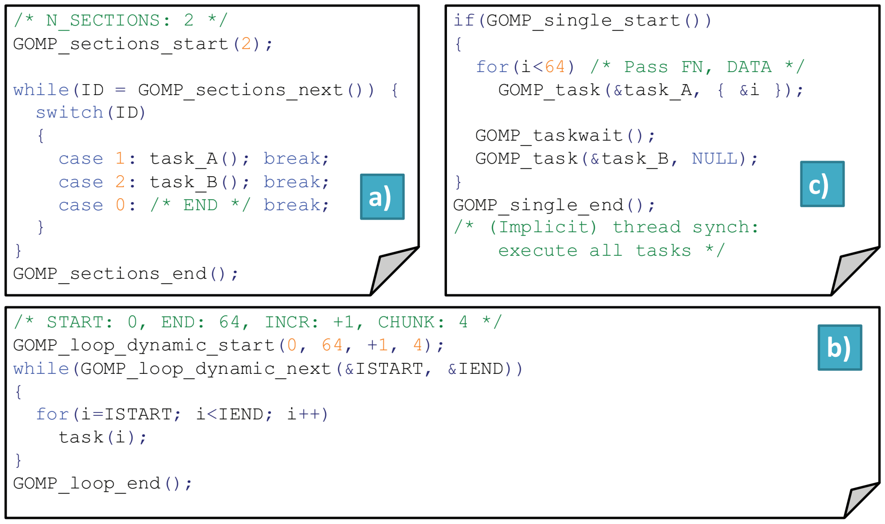
Figure 5.23: Different construct for dynamic parallelism: a) *sections*, b) *dynamic loops*, c) *tasks*

work to be done (e.g., number of iterations, chunk size, global lower and upper bounds of a loop, etc.).

Code snippet in Figure 5.25 shows how the *work-share* data structure is initialized in the `GOMP_loop_dynamic_start()` function, and how the current thread is pointed to the next *work-share* when the loop (or section) is over in the `GOMP_loop_end()` function. These operations can be captured by two generic **INIT** and **END** primitives. Figure 5.26 shows how the `GOMP_loop_dynamic_next()` function updates the *work-share* during *loop* (or *sections*) execution. Figure 5.25 shows also the how the **END** primitive updates thread status (in the `GOMP_loop_end()` function). OpenMP *sections* can be seen as a specialized case of *loops* where $chunk = stride = 1$. A generic **FETCH** primitive can be used to generalize the work-share update operation.

A more in-depth analysis is required for OpenMP tasks.

Figure 5.27 shows execution time breakdown for the **INIT** primitive. The major contributors are the critical region to update the FIFO queue, and the memory allocation for the OpenMP Task descriptor. Since management of pre-allocated memory bins is also a very generic operation, used in virtually every runtime system, this functionality was selected for HW acceleration. The **FETCH** primitive for *tasks* can thus be enriched with this functionalities. A *task* in the work queue can be *executing*, *unexecuted* or *ended*, thus a mechanism for tracing its status must be put in place. To this



```
/* N_SECTIONS: 2 */
GOMP_sections_start(2);

while(ID = GOMP_sections_next()) {
  switch(ID)
  {
    case 1: task_A(); break;
    case 2: task_B(); break;
    case 0: /* END */ break;
  }
}
GOMP_sections_end();
```

```
if(GOMP_single_start())
{
  for(i<64) /* Pass FN, DATA */
    GOMP_task(&task_A, { &i });

  GOMP_taskwait();
  GOMP_task(&task_B, NULL);
}
GOMP_single_end();
/* (Implicit) thread synch:
    execute all tasks */
```

```
/* START: 0, END: 64, INCR: +1, CHUNK: 4 */
GOMP_loop_dynamic_start(0, 64, +1, 4);
while(GOMP_loop_dynamic_next(&ISTART, &IEND))
{
  for(i=ISTART; i<IEND; i++)
    task(i);
}
GOMP_loop_end();
```

Figure 5.24: GCC-transformed dynamic parallelism constructs: a) *sections*, b) *dynamic loops*, c) *tasks*

Type	INIT	FETCH	END	SYNC
Sections	**W** task_descr foreach section update thrd status	**R** task_descr (section_descr)	update thrd status	-
Dynamic loops	**W** loop infos: start,end,.. update thrd status	**R** chunk_istart, chunk_iend	update thrd status	-
Tasking	**W** task_descr	**R** task_desc update task status	update task status	explicit

Table 5.4: Description of the different primitives for each of the three dynamic parallelism constructs

aim the semantics of the **INIT** and **END** primitives were enriched for *tasks*.

Table 5.4 summarizes the functionality of the selected primitives for HW acceleration. Their implementation is discussed in Section 5.6.2.

5.6.2 The Hardware Scheduling Engine

This section describes the *Hardware Scheduling Engine (HWSE)*, a module to accelerate in HW the primitives introduced in Section 5.5.

```
/* INIT */
int GOMP_loop_dynamic_start(int start, int end,
                            int incr, int chunk) {
  gomp_work_share_t ws = /* Init WS */;
  // ...
  ws.chunk = chunk;
  ws.end = ((stride > 0 && start > end)
    || (stride < 0 && start < end)) ? start : end;
  ws.stride = stride;
  ws.next = start;
  return INIT_OK;
}

/* END */
void GOMP_loop_end() {
  current_WS[thread_ID]++;
}
```

Figure 5.25: GOMP code snippet for loop **INIT** and **END**

```
/* FETCH */
int GOMP_loop_dynamic_next (gomp_work_share_t *ws,
                            int * pstart, int * pend) {
  /* 'ws' holds the status on thread's current WorkShare */
  int start, end, chunk, left;
  LOCK();
  start = ws->next;
  if (start == ws->end)
    return WS_ENDED; /* WS finished! */

  chunk = ws->chunk_size * ws->stride;
  left = ws->end - start;
  /* Adjust the boundaries */
  if (ws->stride < 0) {
    if (chunk < left) chunk = left;
  } else {
    if (chunk > left) chunk = left;
  }
  end = start + chunk;
  ws->next = end;
  *pstart = start; *pend = end;
  UNLOCK();
  return WS_HAVE_WORK;
}
```

Figure 5.26: GOMP code snippet for loop **FETCH**

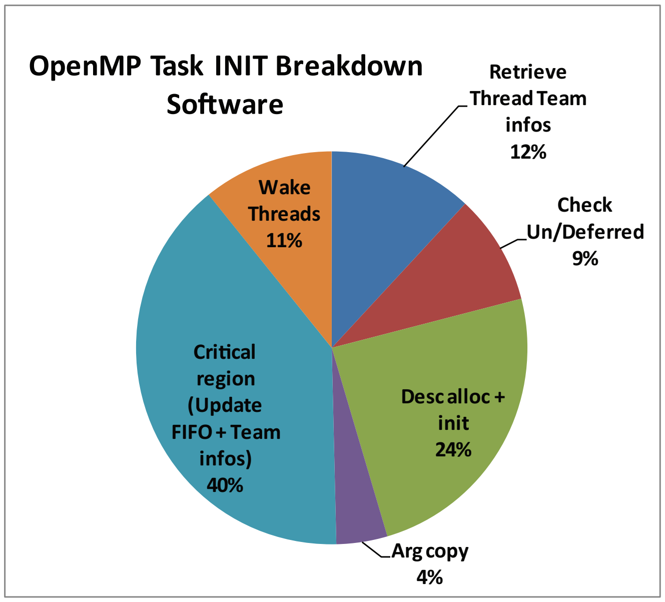


Figure 5.27: Timing overhead of task **INIT** performed in software

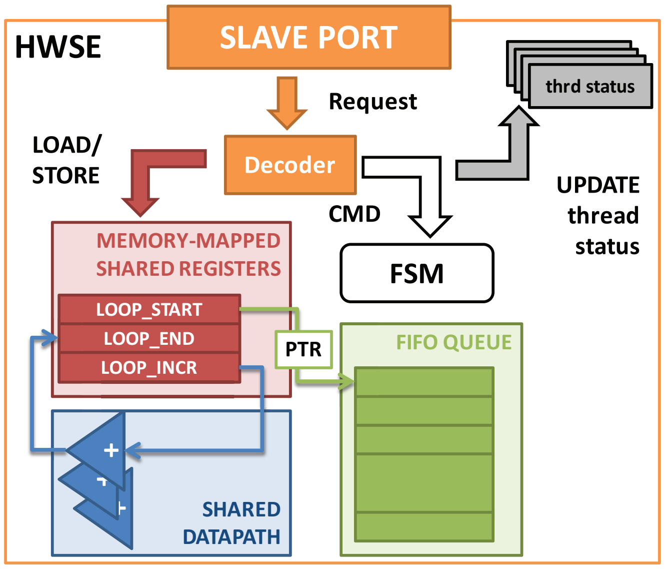


Figure 5.28: Scheme of the HWSE

5.6.3 HW Module Implementation and integration in the cluster

The internal core structure of the HWSE (shown in Figure 5.28) consists of a control finite-state machine that receives the various **INIT**, **FETCH** and **END** primitives and responds accordingly. A central *core* datapath implements these primitives, and has additional logic to specialize their behavior for the construct at hand (*loops*, *sections*, *tasks*, memory allocation). Before using it, the HWSE must be configured to enable the desired construct. This can be done via memory-mapped configuration registers, which are appropriately set within the provided SW routines (`hwse_init_*`, see Section 5.6.1). The **INIT** primitive for *dynamic loops* simply consists of writing lower bound, upper bound and stride into the LOOP_START, LOOP_END, LOOP_INCR registers. The same happens for *sections* (remember they are a special case of loops with *chunk* = 1). Loop boundaries (or the next available section) are computed by a submodule implementing the **FETCH** primitive. A simple circular buffer of 32, 64 or 128 elements implements the FIFO queue; the control FSM is responsible for storing and extracting elements from the queue. Invoking the **END** primitive results in updating a thread-specific register which stores its current *work-share*.

To implement the memory allocator functionality the logic for loop scheduling was entirely reused. In the **INIT** primitive the base address for the memory heap, its global size and the size of a memory bin (containing the specific work/task descriptor) are stored respectively in the LOOP_ START, LOOP_END and LOOP INCR registers. Requests for a new memory bin are serviced through the loop iterations scheduler, until the there are available bins. Then, memory bins are extracted from the FIFO queue (`alloc`) in the **FETCH** primitive, and inserted back therein (`free`) in the **END** primitive.

Task support deserves further discussion. The **INIT** primitive supports the creation of a task (function `GOMP_task()` in Figure 5.24 (c))) by inserting the address of a newly created task descriptor in the FIFO queue. Similarly, the the **FETCH** primitive dequeues a task descriptor address from the queue. The **END** primitive for *tasks* was not accelerated in hardware, and the reason will be explained in next section.

The HWSE is integrated in the target cluster, tightly-coupled to cores through the high-speed interconnection. The FSM can be controlled by the cluster by means of a memory-mapped interface; registers are memory-mapped, and special addresses trigger the different primitives.

A RTL (SystemVerilog) model of the HWSE was implemented, and synthesized using the STMicroelectronics 28 nm bulk low-threshold libraries as a target, with a clock frequency of 400 MHz. Table 5.5 summarizes the results regarding area (in gates)

	Area (kgates)			**Power** (mW)		
#tasks	*32*	*64*	*128*	*32*	*64*	*128*
Decoder FSM	5.42	5.42	5.38	2.42	4.80	9.57
Datapath	3.00	2.81	2.78	1.35	1.27	1.27
Task queue	4.70	9.39	18.69	2.47	2.47	2.46
Total	13.12	17.62	26.85	6.24	8.54	13.30
% of cluster	1.49	1.99	3.01	1.23	1.67	2.59

Table 5.5: HWSE Module

and power (in mW), and the impact on the cluster area and power (in %), which were gathered similarly. Depending on the queue size, the HWSE adds ≈ 1%-3% to the area and power of the original cluster design. Much of the area occupation and power consumption of the HWSE is in the FIFO queue and thus depends of its depth (the maximum number of tasks supported). In absence of contention only 2 (for crossing the IC) + 1 (the delay added by the module) clock cycles are necessary to execute any primitive.

5.6.3.1 Programming Interface and integration in the `libgomp`

To conveniently program the HWSE, a SW API was developed, which abstracts the low-level process of register configuration. Table 5.6 summarizes the functions provided by this API and their description. As already discussed previously, the HWSE implementation is based on the analysis and optimization of the GCC-OpenMP runtime library: `libgomp` (52). Table 5.6 also lists the corresponding functions in the `libgomp` library for supporting dynamic loops, sections and tasks. Starting from this implementation, the schedulers for sections, dynamic loops and tasks were replaced with calls to the HWSE APIs. For the former two constructs the operation was straightforward, due to the one to one correspondence between the HW and SW primitives. Tasks, on the contrary, have much more sophisticated semantics than a simple FIFO queue, which required more work for the integration. Task-level synchronization implies that any thread encountering a `taskwait` construct must *"wait on the completion of child tasks of the current task"* (cit (106)). This implies that parent-children information among tasks must also be stored, other than a FIFO representation. libgomp does so by using a tree data structure. Here, a more lightweight implementation is provided, based on

Type	**HWSE APIs**	**libgomp API**
Sections INIT	`hwse_sections_init_count(n_sections)`	`GOMP_sections_start(n_sections)`
FETCH	`hwse_sections_fetch_ID()`	`GOMP_sections_next()`
Dyn loops INIT	`hwse_loops_init_loop(start,end, incr*chunk)`	`GOMP_dynamic_loop_start(start,end,incr,chunk)`
FETCH	`hwse_loops_fetch_iters(&istart,&iend)`	`GOMP_dynamic_loop_next(&istart,&iend)`
Tasking INIT	`hwse_task_init_desc_addr(addr)`	`GOMP_task(FN_PTR,DATA_PTR,...)`
FETCH	`hwse_task_fetch_desc_addr()`	-

Table 5.6: HWSE APIs

atomic counters (2)(63), handled in software rather than in the HWSE, to maintain the generality of the primitives implementation.

Task descriptors contain information on shared data, thus can become very large. Thus the descriptors themselves are not stored in the HWSE FIFO, but only their address. Descriptors are stored in the shared L1 SPM, so once their address is extracted the SW can quickly access the information therein. As a consequence, the **END** primitive for tasking was not implemented in hardware.

5.6.3.2 Applicability of the HWSE to different programming models

As shown in Table 5.7 , most programming models are implicitly asynchronous, and in some cases also embody a fork-join execution model. The semantic of **INIT/FETCH** primitives perfectly matches the behavior of a work queue supporting asynchronous execution. Synchronicity (and synchronization) can be implemented with the support of the **FETCH** primitive, e.g., wrapping it in a software loop until there are tasks to execute. Complex dependencies between tasks (e.g., parent-children relationship) can be expressed enriching the descriptors of the work to execute. Indeed, the primitives (and the corresponding HWSE implementation) agnostically handle a language-specific data structure describing a single task, that therefore can include information – such as references to other task structures – to be managed by a higher software level. The work descriptor can be enriched also to specify a set of tasks, e.g., to support data parallelism similarly to what happen in Intel TBB (63), where high-level data parallel constructs are built on top of a task scheduling library. Finally, all the programming models shown in Table 5.7 abstract memory allocation to software. The HWSE proposed in this work can be configured as a pre-allocated memory manager to support and accelerate memory allocation and free primitives.

Name	Explicit a/synch exec.	Task synchronization	Task Parallelism	Data Parallelism	Inter-task dep.	Memory allocation
Intel TBB (63)	Fork/Join	Explicit Join	Tasks	Lib built on top of Task scheduler	Group `spawn_and_wait`	Implicit in Class Inherit.
OpenMP (106)	. Implicitly asynch . Explicit synch	. Implicit at TSP . Explicit (`taskwait`)	*Tasks* *Sections*	*Dynamic Loops*	Parent-child `taskwait`	Transparent
Cilk (96)	Fork/Join	Explicit Join	*Co-operative* Tasks	`cilk_for`	-	Transparent
Apple GCD (7)	Synch/Asynch (queue-based)	Explicit (Q WAIT)	Tasks	-	-	Implicit in Q ALLOC
Plurality CSU (109)	Asynch	Token-based	*Regular* Tasks	*Duplicable* Tasks	Tokens	Transparent

Table 5.7: Most relevant programming models supporting dynamic parallelism

5.6.4 Experiments

The proposed cluster was prototyped using a SystemC Virtual Platform (26) which models the HWSE integrated in the cluster platform described in Section 5.3, with main architectural parameters as summarizes in Table 5.9. With this setup, the approach was validated both with synthetic benchmarks, and applications from image processing domains.

5.6.4.1 Synthetic benchmarks

The first experiment to measure the performance improvement brought by the HWSE compared to the software schedulers consists of three synthetic workloads. To test accelerated *sections*, 24 sections were spawned each consisting of 100 NOPs (to prevent side effects due to memory contention). For *dynamic loops*, a loop was created of 64

Type	**JPEG**	**Tracking**	**Strassen**	**FAST**	**Face Detection**	**Average**
	GCC-OpenMP (52)					
Loops	**27%**	**6%**	*S1*: **80%** *S2*: **28%** *S3*: **80%**	**53%**	**20%**	**42%**
Tasks	**26%**	**27%**	*S1*: **26%** *S2*: **0.5%** *S3*: **22%**	**48%**	**3%**	**21.8%**
	OMPI (2)					
Loops	**48%**	**27%**	*S1*: **83%** *S2*: **83%** *S3*: **82%**	**83%**	**85%**	**70.1%**
Tasks	**80%**	**97%**	*S1*: **96%** *S2*: **44%** *S3*: **97%**	**95%**	**90%**	**85.6%**

Table 5.8: HWSE performance improvement against libgomp and OMPi SW schedulers.

ARM v6 cores	16	# L1 SPM banks	32 (K=2)
L1 SPM size	256 KB	# L1 SPM latency	$\geq$ 2 cycles
L3 size	256 MB	L3 latency	$\geq$ 59 cycles
$I\$_i$ size	1 KB	$I\$_i$ line	4 words
t_{hit}	= 1 cycle	t_{miss}	$\geq$ 59 cycles

Table 5.9: Architectural parameters

iterations each containing 100 NOPs, while for the *tasking* 18 tasks were spawned each containing a loop of 5000 iterations of 100 NOPs. All the processors are involved in the computation. Table 5.10 summarizes the speedup brought by the HWSE for each of the three primitives, over the pure SW version.

Type	INIT	FETCH	END
Sections	16×	78×	181×
Dynamic loops	1.07×	6×	14×
Tasks	1.41×	1.21×	-

Table 5.10: Sections and loop speedup compared to the pure SW version

Accelerated *sections* provide the best speedups, significantly higher than *dynamic loops* even if the two constructs share almost identical semantics. The reason for this difference is that each **INIT** and **FETCH** event for the *sections* implies a single write in the HWSE, while *loops* require multiple consecutive writes. Consequently the HWSE must be locked to prevent non-mutually exclusive updates from distinct threads. This operation is done in software, and implies the difference in performance that we can observe. Similarly, the HWSE can only accelerate a portion of the sophisticated task scheduler, leaving a relevant portion of the code to be executed in software. For this reason we observe a more modest 41% speedup for the **INIT** and 21% for the **FETCH**. The **END** primitive, as already explained, was not accelerated.

5.6.4.2 Comparison with software schedulers

The HWSE was compared against two freely available OpenMP runtimes, namely `libgomp` (52) and OMPi (2). Both runtime systems have been ported on the target cluster platform. For this comparison 5 image processing applications were considered: JPEG decoding, Color Tracking, Strassen matrix multiplication, FAST corner detection, Viola-Jones Face Detection. For each of them were proposed, where possible, two alternative implementations: one which uses *tasks* and one which uses *dynamic loops* or *sections*. In both cases work units were generated as fine-grained as possible, to verify the effectiveness of the HWSE.

Table 5.8 shows the performance improvement for each application, when the HWSE is compared to the software schedulers in `libgomp` and OMPi. For the Strassen matrix multiplication the speedup for each of the three main phases of the algorithm is plotted. We see almost no performance gain for stage 2, because the work units are very coarse grained, which tends to minimize the impact of the software overheads A similar situation takes place for the task-based version of Face Detection. Besides these two cases, on average the HWSE achieves $\approx 32\%$ speedup versus libgomp, and $\approx 76\%$ speedup versus OMPi.

5.7 Conclusions

Many-core clustered architectures are increasingly being adopted to design embedded many-cores. These platforms can deliver very high peak performance within a contained power envelope, provided that programmers can make effective use the available parallel cores. This is becoming an extremely difficult task, as embedded applications are growing in complexity and exhibit patterns of parallelism that is multi-level, and/or irregular in nature. To cope with the increasing application complexity, several languages and extensions were proposed, one of the major being OpenMP. OpenMP lets the programmer specifying complex forms of parallelism in its code, such as *nested* and irregular (*tasking*) parallelism, but, to effectively support them on modern many-cores, a runtime support is required, and it must be designed *ad-hoc* for efficiently exploiting the underlying hardware. In this chapter, a novel design was proposed for the GCC-OpenMP runtime (`libgomp` (52)), which supports both *nested* and *tasking* parallelism. The runtime support for *nested* parallelism was validated against a runtime whose design is capable of simple flat parallelism, showing how the introduced overhead is negligible compared to the achievable performance gain.

For the *tasking* support, several architectural variants were considered for the target cluster, aimed at removing the bottlenecks arising in the shared memory system. All of them were exhaustively validated, on a virtual platform using both synthetic benchmark and real applications from the image processing domains. Performance of the runtime was characterized in terms of the minimum granularity of tasks than can be effectively spawned on it, outperforming existing implementations by a factor of 20×.

At last, key functionalities of the `libgomp` were identified and a hardware implementation for them was proposed, based on a Hardware Scheduling Engine (HWSE) that was tightly-coupled inside the cluster template. The performance gain given by the HWSE was validated with several benchmarks running on a virtual platform simulation, and its area and power estimated with RTL synthesis. Results prove the effectiveness of the proposed approach.

6

Heterogeneous many-core clusters with shared memory

Modern embedded systems increasingly adopt heterogeneous platforms, where hardware accelerators are coupled to cores, to trade specialization of the platform to a specific application domain for speedup and energy efficiency. In a shared-memory platform, the task of communicating and synchronizing between hardware accelerators and cores can be achieved *via* shared memory banks, so that the data copy needed – e.g., in a GPU-based system – to feed the accelerators are completely removed. This is called *zero-copy* scheme (5). This chapter targets this kind of architecture, and three main issues are considered, namely *platform design*, *architectural scalability* and *programmability*. This part of the thesis is the result of a cooperation between University of Bologna and Université de Bretagne-Sud, under a joint PhD agreement.

6.1 Introduction

Modern embedded systems are increasingly exploiting architectural heterogeneity to improve energy efficiency and performance. High-end products such as smartphones and tablets typically include hardware accelerators, and we can foresee that this trend will continue, possibly exploiting on-chip programmable logic (142) to customize the system functionality to a specific application domain. This happens especially for image and video processing systems, which greatly benefit from hardware acceleration of critical computation-intensive code *kernels* such as convolutions, discrete cosine transform, color space conversion etc., frequently found at the heart of several applications. These key *kernels* are implemented in ASIC/FPGA technologies, and the corresponding hardware blocks are coupled to general-purpose cores as co-processors. This form of architectural heterogeneity is an effective solution to improve energy efficiency of SoC designs by a factor of 10× to 100×.

Key to designing such heterogeneous platforms are efficient mechanism for core-to-accelerators communication and synchronization. In shared-memory systems, such as the ones considered in this work, these tasks can be easily implemented through shared memory banks. This clean and efficient communication scheme greatly simplifies the process of designing such a platform and the integration of hardware accelerators with the software stack. The target architecture here considered are shared-memory many-core heterogeneous clusters shown in Section 2.1, and more in details, the proposal is to tightly-couple accelerators inside the clusters, so that they share the L1 memory banks with the general purpose cores. This is called *zero-copy* scheme, and from now on we will call the accelerators implementing it Hardware Processing Units - HWPUs. The template for the target heterogeneous cluster is shown in Figure 6.1.

In such an template, there is no need to perform data movements other than the ones which are however necessary to increase the locality of data, as opposite to what happens e.g., in GPU-based systems, where data movements are necessary (e.g., using OpenCL (80)), and efficiently orchestrating them is paramount to achieving performance.

The architectural template and communication scheme are clearly defined, and this enables the development of design techniques and tools, and a modular design for the software stack to support programming. The *zero-copy* scheme perfectly fits the semantics of most of the programming models used in embedded systems (e.g., C, C++), which are shared-memory based.

Once the communication scheme have been fixed, there are three challenges in designing heterogeneous many-core platforms:

1. *Platform design.* This means providing methodologies and tools to help engineers in prototyping and exploring the design space of such an architecture and HW/SW partitioning of an application. Roughly speaking, this means answering to questions such as: "How can I identify the portions of an application that

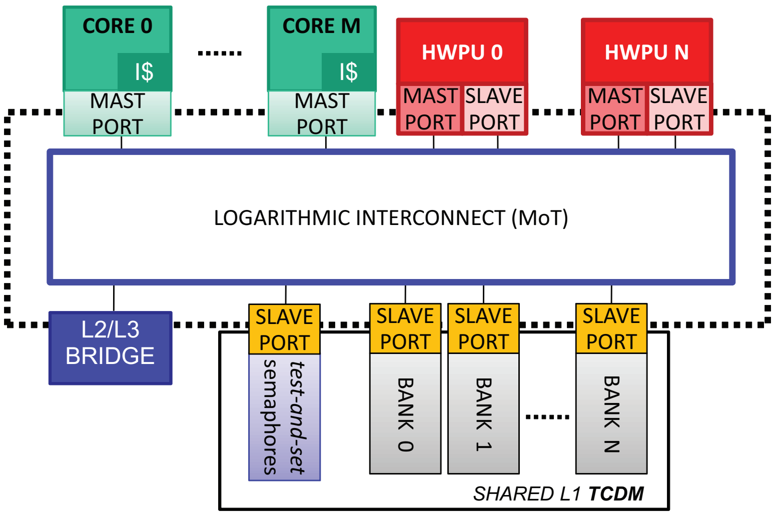


Figure 6.1: Proposal for an heterogeneous cluster with shared-memory

should be accelerated on HW rather than in SW?", "How many HWPUs and cores should I plug in the cluster?".

2. *Architectural scalability*, that is, providing architectural templates to include several (tens of) HWPUs inside a many-core cluster, identifying and solving the bottleneck that may arise (e.g., the memory bandwidth) when doing so.

3. *Programmability*. Software developers must be provided with compilers, runtime libraries and programming languages to efficiently exploit HWPUs from within their (already existing?) code, and develop applications that are modular, portable and scalable (in the "software engineering" meaning of the word).

While considering these problems, there are two possible "points of view", namely a *top-down* approach, where platform is designed from scratch given a high-level specification of the *kernels* to accelerate (e.g., in C language), and a *bottom-up* approach, where accelerators that already exist in the platform must be detected and exploited from application code. This chapter explores both of them. The programming model considered is OpenMP, for which at the time we write this thesis, the steering board is discussing (specifications 4.0 (107)) future extensions to cope with hardware acceleration. Here, a few custom APIs are proposed, that achieve this goal.

6.2 Related works

SoC architectures featuring HW accelerators have been widely studied in literature. Approaches based on ASIP and Instruction Set Extensions (ISE) exhibit a limited data bandwidth between processors and acceleration logic due to constraints on register file port availability. To overcome this limitation, Architecturally Visible Storage (AVS) (20) uses local memory elements to intrinsically increase the data bandwidth. Allowing separate L1 memory segments introduces consistency issues when several copies of data are created. In particular, if processors leverage L1 data caches AVS suffers from coherence problems. Kluter et al. (78) present a memory coherence scheme for ISEs with AVS meant to ensure execution correctness with minimal area overhead. An evolution of this approach, *Way Stealing*, is presented in (79), where AVS is implemented by locking target lines in processors' caches and is coherent by construction. AMD Fusion provides a GPU-based architecture which implements *zero-copy* communication, and to the best of my knowledge, they are the first adopting this terminology. It is shared-memory GPU-based architecture, where the accelerator unit (APU) is not tightly coupled to cores, and hardware tasks are dispatched through a FIFO queue system. Fajardo et al (50) propose an architecture where cores and accelerators share a common L2 SRAM, called Buffer-Integrated-Cache (BIC). However, accelerators are not coupled at L1, hence *"the shared BIC cannot completely eliminate physical local storage"*. Moreover, they add what they call a *BIC substrate*, to let the cores access it as a cache, and accelerators access it as a local storage (but with higher latency). Their design is more complex and less efficient than the proposed heterogeneous clusters and – most important – not scalable to a high number of accelerators, because the BIC substrate is implemented as a shared *"augmented conventional cache"*.

Some works attack the scalability problem by providing smart controllers for the accelerators that resemble the ones proposed here. Bin et al. (86) propose a Memory Access Engine (MAE) to hide the increasing latency towards the shared memory system as the number of PEs grow. To do so, they perform data prefetching based on access patterns that are recurring in the image processing domain. However, their accelerators are not tightly-coupled neither to cores nor to the memory banks, thus the *zero-copy* scheme cannot be implemented efficiently.

The approach of Cong et al. (37) is probably the closest to this work from the point of view of programmability. They propose a hardware module (called GAM) for supporting the execution of accelerators, to tackle programmability issues. The GAM features hardware support for virtualizing and *composing* the available accelerators, to offload complex units of works (*macro-tasks*). However, their architecture features loosely-coupled accelerators, thus it significantly differs from heterogeneous clusters. Each accelerator resides on a different tile, other than cores and memory banks, so there are no opportunities for efficiently implementing shared memory-based communication.

Conservation-Cores (118, 135) provide a generic template for building low-power, reconfigurable accelerators. However, they are not application-specific circuits but general cores, whose goal is not on improving performance but energy-delay.

Programming models will be key instruments to efficiently program accelerator-based MPSoCs. Several efforts in academia (8, 28, 37) and industry (65) are pushing for solutions to ease programmability on accelerator-based platforms, as witnessed by initiatives such as HSA (82) and Khronos (127). The Khronos vision working group has been formed to drive industry consensus to create a cross-platform API standard (OpenVX (128)) that enables hardware vendors to implement and optimize accelerated computer vision algorithms. The Khronos vision API can accelerate high-level libraries, such as the popular OpenCV open source vision library (104), or can be used by applications directly. The Khronos vision working group has not converged towards a standard for OpenVX yet, but Section 6.5 shows how embracing this "philosophy" is highly beneficial on the considered platform.

OpenCL (80) attempts to unify the programming models for such platforms into a unique standard. However its low-level programming style requires deep knowledge of the underlying platform, making it cumbersome to use for non-experts. The more appealing directive-based approach of OpenMP inspired a number of other approaches, such as PGI Accelerator (130) or StarSS (9). OpenMP has also been proposed as a programming model for FPGA-based accelerators (29), or to co-specify a HW/SW platform including multiple cores and accelerators (60). Both the architectures leverage data copies to feed accelerators.

Several methodologies and tools exist to simplify the process of designing hardware accelerators from high-level descriptions. In the context of ASIP acceleration, Synopsys (120) and Tensilica (122) provide complete environments and toolchains for the development and testing of application specific processors and software. OpenMP has been proposed as a high-level interface to describing hardware in (13, 43, 85), typically targeting FPGA/ASIC accelerators. Different from these proposals, the way OpenMP is used here aims at defining a complete HW/SW architecture, where multiple cores and accelerators communicate through pre-defined interfaces.

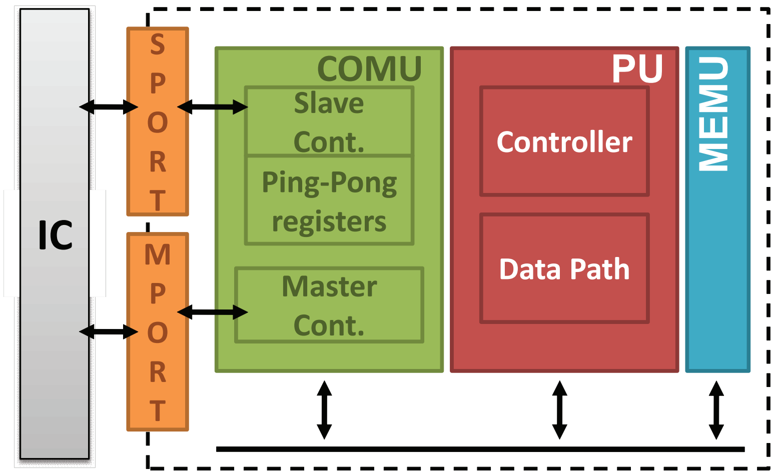


Figure 6.2: Structure of a HWPU

6.3 HWPUs: *zero-copy* tightly-coupled hardware accelerators

The Hardware Processing Units (HWPUs) are user-defined hardware accelerators that are tightly-coupled inside a cluster, and embody the *zero-copy* communication scheme. Different from the typical host+accelerator architecture, in the *zero-copy* accelerator model data need not to be moved in and out of the accelerator private memory space. In a cluster such as the one depicted in Figure 6.1, HWPUs access data directly from the L1 scratchpad through the logarithmic interconnect. This has a key advantage. While – similar to traditional NUMA systems – in a cluster-based architecture moving data to local memories is crucial to achieving performance, once a given data item (e.g., a slice of a large image) has been brought close to the processors (e.g., the considered TCDM, or a data cache in a different architecture), no further movements are required to make it visible to HWPUs. The HWPU template is composed of three main functional units: a processing unit (PU), a memory unit ($MEMU$) and a Communication Unit ($COMU$)

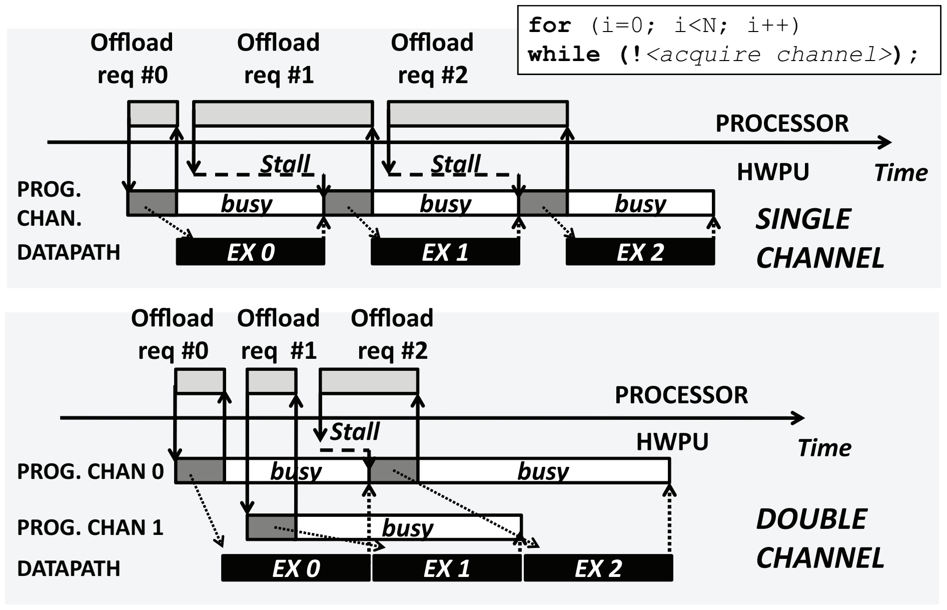


Figure 6.3: Timing diagram for programming HWPU channels

(see Figure 6.2). The *PU* is a data-path providing the implementation of the target algorithm. The *MEMU* is composed of memory banks to store temporary results, coefficient values, etc. The *COMU* includes in/out buffers, a synchronization processor and a set of registers, used to initiate an offloading sequence from a processor. These registers (see Table 6.1) are mapped in the global address space, thus HWPUs can be programmed by directly writing therein. A duplicated set of such registers is provided,

Register name	SIZE	R/W	Brief description
`working`	32	R	1 = HWPU doing work
`in_addrs`	8×32	W	In parameter addresses
`out_addrs`	8×32	W	Out parameter addresses
`trigger`	32	W	Starts HWPU execution

Table 6.1: Memory-mapped registers to interact with a HWPU

which can be seen as a *double programming channel* to schedule jobs on the HWPU. Two processors concurrently trying to offload a job on the same HWPU do not need to wait for the competing processor to complete its programming sequence.

Figure 6.3 shows a timing diagram of the offload and execution sequence when single and double programming channels are used to program a HWPU. Double programming channel allows to overlap the execution phase for an offload request with the programming phase of another. Note that double channels only involve the programming phase of the HWPU, and they are not to be mistaken for *double buffering* techniques to overlap execution and data transfer. The two mechanisms are orthogonal one another, and can be adopted at the same time.

When the HWPU needs to access data from memory the corresponding transaction is appropriately packetized to match the interconnect protocol and brought to the master port (MPORT). The interconnect supports 32bit-wide transactions. To enable higher data bandwidth, the generic HWPU template supports multiple master ports. Access requests to contiguous memory addresses are thus split into multiple parallel transactions on different ports. The semantics of accelerator execution is *non-blocking*: once a processor has successfully completed the set of steps required to offload a given code kernel, it can asynchronously execute independent code, without the need to wait for the accelerator to finish. Synchronization can be enforced at specific points by checking for HWPU termination on the `working` register.

When a processor wants to offload a job on the HWPU it initiates a programming sequence which consists of the following steps:

1. it acquires a lock on the available programming channel

2. it notifies the addresses of each input and output data item in the `in_addrs` and `out_addrs` sets of registers

3. it triggers the HWPU by writing on the `trigger` register.

4. upon encountering a synchronization instruction it busy waits on the `working` register.

A possible programming sequence implementation, where offloading is aborted in case there are no available channels is shown in the following code listing.

```
int
hwpu_program (unsigned int hwpu_ID, int numin, int numout,
              unsigned int *inaddr, unsigned int *outaddr)
{
  unsigned int i;

  WAIT (LOCKOF (hwpu_ID));
  /* Acquired the lock. Start programming sequence */

  if (!hwpu_has_free_slot_1(hwpu_ID))
  {
     /* HWPU has no free channels. Offload failed. */
    SIGNAL (LOCKOF (hwpu_ID));
    return HWPU_OFFLOAD_FAILED;
  }

  hwpu_start_prog (hwpu_ID);

  hwpu_write_indatacount (hwpu_ID, numin);
  for(i=0; i<numin; i++)
    hwpu_write_inaddr (hwpu_ID, inaddr[i]);

  hwpu_write_outdatacount (hwpu_ID, numout);
  for(i=0; i<numout; i++)
    hwpu_write_outaddr (hwpu_ID, outaddr[i]);

  /* We loaded in/out data addresses. Trigger the HWPU. */
  hwpu_trigger (hwpu_ID);

  /* Release the HWPU, so that other cores can program it. */
  SIGNAL (LOCKOF (hwpu_ID));

  return HWPU_OFFLOAD_OK;
}
```

The following listing shows an implementation of the `hwpu_wait`, which continuously polls on its `working` register.

```
void
hwpu_wait (unsigned int hwpu_ID)
{
  while (hwpu_read_reg (hwpu_ID, HWPU_WORKING_REG) == TRUE)
    ;
}
```

6.4 Top-down approach: HLS-based flow for platform design

To quickly build and explore several HW and SW architectures, this section describes an integrated design flow which starting from the high-level OpenMP application specification automatically generates a cycle-accurate simulation model of the target platform. At first, an extension to OpenMP is proposed, to annotate portions of code suitable for acceleration in HW, then the automated design flow is shown. The adoption of OpenMP enables a mix of parallelization and acceleration. A modified GCC compiler outlines annotated code regions into synthesizable C code - from which a SystemC model of the HWPU is generated - and substitutes HW-accelerated code segments in the original application with a sequence of instructions to offload computation to the target HWPU. The adoption of OpenMP enables a mix of parallelization and acceleration using the same programming model. Then, the HWPU accelerator model is compared against ASIP solutions and traditional accelerators with copyin/copyout semantics. Experimental results confirm that a synergistic approach based on a mix of parallelization and acceleration provides excellent performance and scalability.

6.4.1 OpenMP extensions for hardware acceleration

OpenMP provides a key construct, the `#pragma omp parallel` directive, to specify code regions that must be executed concurrently by parallel threads. Worksharing constructs are provided to partition the workload within those code regions among threads, and allow programmers to control how application tasks are to be mapped on hardware computational resources. The set of worksharing constructs was extended with a key custom `#pragma omp accelerate` directive to outline code regions which are to be hardware-accelerated, rather than executed in software on some processors.

```
        #pragma omp accelerate [clause[[,]clause]...]
            structured-block
```

where *clause* is one of the following:
`private (`*`list`*`)`
`firstprivate (`*`list`*`)`
`lastprivate (`*`list`*`)`
`num_hwpus (`*`integer-expression`*`)`

```
nowait
```

Data items referenced within an `accelerate` region can be made private by annotating it with the `private` clause, which is also the default datasharing attribute. `firstprivate` variables are initialized with the value of the twin variable from the enclosing parallel construct at the beginning of an accelerate region. `lastprivate` variables update the value of the twin variable with theirs at the end of the accelerate region. `shared` variables implement both these mechanism. Roughly speaking, they respectively specify a variable as input, output and inout data of the accelerators. The `num_hwpus` clause allows to specify how many HWPUs of the same kind are to be created to accelerate the target code region. By default, upon encountering an `accelerate` region the processor is stalled until the offloaded (hardware) task returns. The HWPU can be programmed to work asynchronously with the processor by using the `nowait` clause. In this case, a custom `omp_wait_HWPU` library function can be manually inserted in the code at the point where is necessary to enforce a synchronization point.

The HW accelerator template is seamlessly integrated in the OpenMP execution and memory model. From the point of view of the programming model, an accelerated code region is transparently abstracted as a generic task that performs best if mapped on a specific HW resource, the target HWPU. At the system level, the latter is treated as a generic computational resource communicating through the main shared memory. In respect of the OpenMP relaxed consistency memory model (see Chapter 2), HWPUs are allowed their own temporary view of shared variables (into local buffers), but upon synchronization a consistent (system-wide) view of the shared memory is enforced. Among other advantages, this accelerator model simplifies application development, since the programmer sees a unique memory space and needs not take responsibility for explicitly moving data across distinct address spaces.

Let us consider the example in the code snippet below.

```
        #pragma omp parallel shared (a) private (i)
        #pragma omp accelerate shared (a) private (i)
        for (i=0; i<N; i++)
          a[i] = .. a[i] ..;
```

Here the array `a` is read and written within a loop, which is annotated for acceleration. The array is declared as `shared`. The compiler transforms the `accelerate` construct as follows:

```
int *in_addrs, *out_addrs;

in_addrs = { &a };
out_addrs = { &a };
/* 11 is the ID of HWPU */
while (omp_program_HWPU (11, 1, 1, in_addrs, out_addrs)
       != HWPU_OFFLOAD_OK);
omp_wait_HWPU (11);
```

The compiler infers the presence of one input data item and one output data item within the accelerated region, which addresses are stored in the `in_addrs` and `out_addrs` compiler-generated temporaries. To actually trigger the offloading of the task a call to the `omp_program_HWPU` function is emitted. This function implements the standard HWPU programming procedure described in Section 6.3. It is implemented as an extension to the OpenMP runtime library, and takes five parameters: a unique ID to identify the target HWPU, the number of variables read (input) and written (output), and the two arrays containing their addresses. It is assumed that the processor stalls if none of the programming channels is free. It thus becomes the software's responsibility to make sure that a query to the HWPU is successful. To implement this semantics, the compiler nests the call to the programming primitive `omp_program_HWPU` within a `while` constructs. The program can continue beyond the `while` construct once offloading returns successfully. By default synchronous offloading is assumed, thus a call to the `omp_wait_HWPU` synchronization primitive is automatically inserted at the end of an accelerated region, unless the `nowait` clause is specified.

At the moment this document is being written, new OpenMP specifications (4.0 (107)) are being published that expose the concept of *execution device*, to the software layer. This is clearly introduced to shift to a more "device-oriented" model of execution, to efficiently exploit hardware acceleration opportunities. Exploring the applicability of the new OpenMP 4.0 to heterogeneous cluster is hence a promising direction for future research.

6.4.2 Toolflow for platform design

The process of devising a HW/SW co-design for application acceleration typically involves several steps. First, once candidate program code segments for acceleration have been identified they must be manually extracted from the application, and partly rewritten/adjusted to match a format understood by High-Level Synthesis tools. Second, it is necessary to define a clear interface to allow HW and SW components to exchange data and synchronize correctly. Third, the application code must be transformed so as

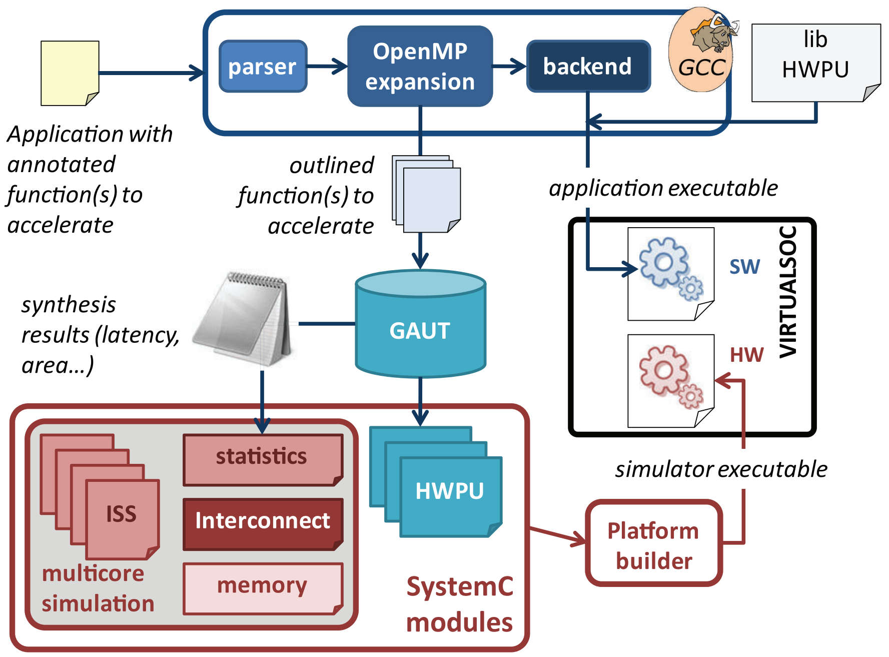


Figure 6.4: Tool Flow

to replace original SW execution of the target kernels with their HW counterparts (i.e., by offloading code to the HWPUs). Here, a complete toolflow is described that simplifies the process of building all the necessary HW and SW components for design space exploration of accelerator-based architectures. Figure 6.4 shows a pictorial overview of the design flow, depicting all the involved steps and interfaces. The target test platform is based on cycle-accurate SystemC simulation (26) of the several key architectural blocks described in Section 6.1. Programmers specify application partitioning (parallel and accelerated code regions) at a very high level of abstraction by using the enhanced OpenMP programming model introduced in Section 6.4.1. The GCC OpenMP compiler (52) was modified to manipulate the program in a two-fold way. The annotated code regions for acceleration are automatically outlined into a suitable file format for the HLS tool to operate on it. Outlined functions are replaced in the application code with library functions that trigger code offloading. An executable image of the transformed application is the final output of this compilation step. The HLS tool automatically extracts the functional SystemC description of the HWPU, plus a VHDL model. The latter is an input for logic synthesis tools, from which timing (and area) information are gathered. HWPU execution latency (processor cycles) information is integrated in a dedicated module for gathering *statistics* (performance, energy) in the simulator, and is used to evaluate correct execution time spent on the accelerator. The described design

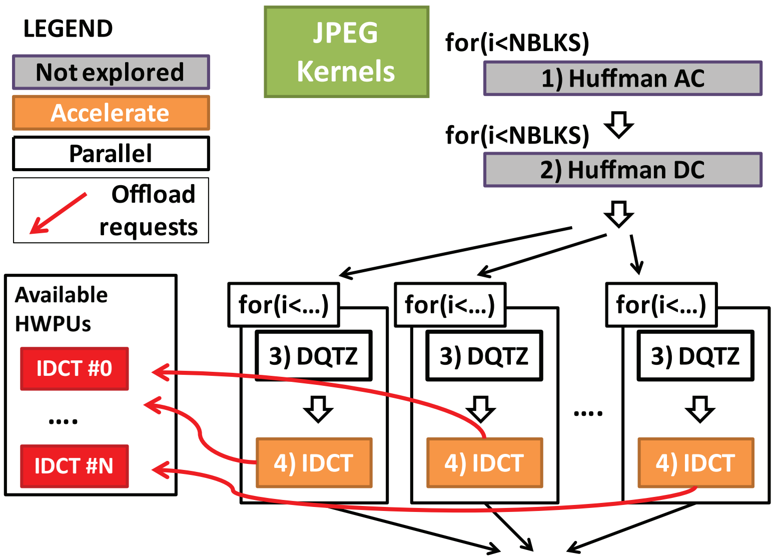


Figure 6.5: JPEG benchmark partitioning

flow leverages a research HLS tool, GAUT (133), but any other research or commercial tool could be integrated. Similarly, other components of the flow could be replaced or integrated with different tools. The overall approach will still work with just minor modifications. The input specification to GAUT has to be written in C and should not include any notion of time or explicit parallelism. Constructs that are not supported for synthesis mainly include dynamic allocation (malloc, free), pointer arithmetic and dynamic loops.

Required parameters to specify the target technology (component library) and the clock period can be passed to HLS tool in a text file, and in the future it will be possible to directly pass them to the compiler by means of custom flags. Additional flags are provided to optionally control interface synthesis, array to memory mappings, hardware hierarchy through function calls, scheduling (latency/cycle) constraints, allocation directives to constrain the number and/or type of hardware resources, etc. As a last step of the tool flow, all of the SystemC modules are assembled into the Virtual-SoC MPSoC simulator (26), which is capable of executing the accelerated application. This integrated approach allows very fast yet accurate definition and implementation of several accelerator-based architectures.

ARM v6 cores	16	TCDM banks	16 (K=1)
$I\$_i$ size	1 KB	TCDM size	256 KB
$I\$_i$ line	4 words	L3 latency	100 cycles
t_{hit}	= 1 cycle	L3 size	256 MB
t_{miss}	$\geq$ 59 cycles		

Table 6.2: Architectural parameters

6.4.3 Experimental Results

This section evaluates an instance of the target cluster presented in Section 6.1 (Figure 6.1). Architectural parameters are summarized in Table 6.2.

A set of kernels from two real applications is considered: a JPEG decoder and a Scale-Invariant Feature Transform (SIFT), a widely adopted algorithm in image recognition systems. By profiling the execution cycles spent over different kernels, the most time-consuming ones were identified and sped-up by means of parallelization, acceleration, or a mix of the two.

For JPEG the focus is on the *dequantization* (DQTZ) and on the *inverse DCT* (IDCT) kernels. DQTZ is parallelized, IDCT is accelerated. From the SIFT algorithm three kernels are extracted: image up/down-sampling (SMPL), Gaussian Filtering (GAUS) and Difference of Gaussians (DOG). GAUS is accelerated while SMPL and DOG are parallelized. Figures 6.5 and 6.6 show a block diagram representation of the two target applications considered in the experiments. Figures highlight the kernels used in the experiments, and the workload partitioning strategy. White blocks represent parallelized kernels, orange blocks represent hardware-accelerated kernels, gray blocks represent kernels not considered in the experiments.

Three different sets of experiments are performed:

1. Comparison of the Zero-Copy HWPU model with other traditional acceleration strategies;

2. Explore how sharing a single HWPU among an increasing number of cores scales, and how double channels help;

3. Explore the effect of multiple identical HWPUs.

Since the focus is on the effectiveness of zero-copy processor/HWPU communication at L1 as compared to other approaches, experiments do not account for those data transfers from main memory to the TCDM required in every acceleration solution to

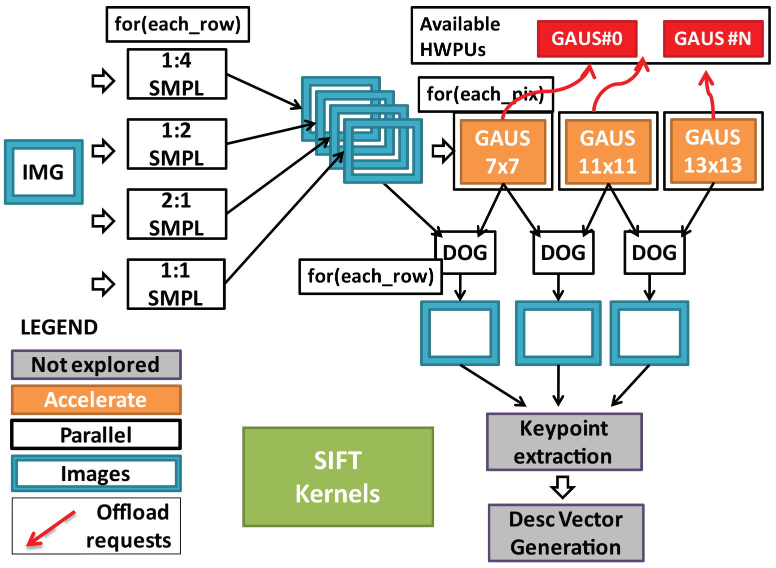


Figure 6.6: SIFT benchmark partitioning

keep most frequently accessed data in L1. More precisely, the snapshot of the system only captures the effects of program execution after data has been brought in L1 and before it is flushed back to main memory. Transfers to/from L2 or L3 are only modeled for those copy-based acceleration approaches where communication can only take place during the program through that particular memory level. Data transfers to/from main memory to deal with limited L1 size are not explored.

6.4.3.1 Comparison of Acceleration Techniques

In this section the *zero-copy* HWPU model (*ZC-HWPU*) is compared against typical acceleration approaches.

Standard ASIP can be considered as an acceleration technique where ALU and acceleration extension logic share L0 memory (i.e., the register file). The number of read and write ports to the register file poses a constraint on the number of input and output operands of the accelerated function (typically 2 inputs, 1 output). Thus, even if standard ASIP provide the tightest coupling to a “standard core”, efficiently accelerating computation of arbitrary complexity, the footprint of the involved dataset is necessarily very small. To model ASIP acceleration clusters of instructions were identified in the disassembled application code which operate on two inputs and produce a single output, and replaced with a single special instruction executing in one processor cycle.

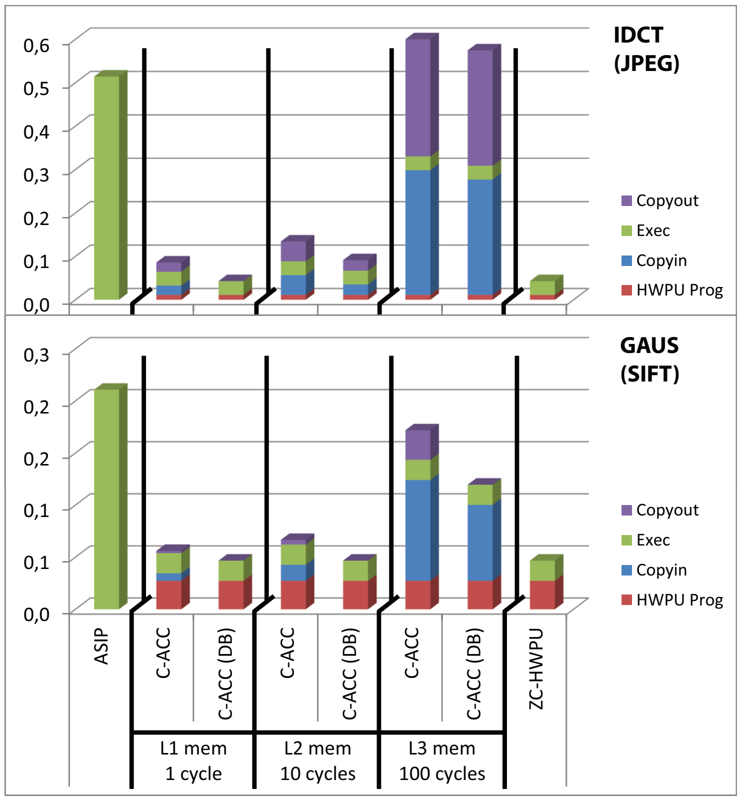


Figure 6.7: Comparison of different acceleration strategies.

Another traditional acceleration approach is one where accelerators access input/output data from a private L1 memory (a buffer, or a scratchpad). Here, control processors are responsible for moving data back and forth from main memory to private accelerators memory. Enhanced ASIP techniques such as Architecturally Visible Storage - AVS (20) fall in this category. In the following we will refer to this acceleration approach as *C-ACC*. *C-ACC* accelerators have same datapath generated for *ZC-HWPU* (i.e. the same execution cycles count), but memory accesses are not injected into the system interconnect, rather all memory requests are satisfied from a local scratchpad with 1-cycle latency. Data transfers from main memory are modeled as 4-word DMA bursts. All of the HWPU implementation have 4 MPORTs. Information on latency is extracted from synthesis targeting FPGA.

Tables 6.3 and 6.4 summarizes the synthesis results (target *Xilinix V5LX110T* FPGA) for the two accelerated kernels, namely IDCT and GAUS. Area (slices), latency (cycles) and speed (frequency) results are provided for three HWPU implementation featuring 1, 2 and 4 MPORTs respectively, to explore the tradeoff between area cost

	Data Footprint: INPUT=256B, OUTPUT=256B							
	Operands				Reg	Area	Lat.	Freq
	add	mul	sub	sra	(1bit FF)	(slices)	(cycles)	(MHz)
Parallelism 1	10	8	4	3	5582	15499	178	218
(1 MPORT)	4	3	2	1	4583	12580	368	200
	2	2	1	1	5730	11485	628	213
Parallelism 2	14	13	7	3	6463	18028	114	212
(2 MPORTs)	4	4	3	1	5542	15840	304	216
	3	2	2	1	5756	13023	564	219
Parallelism 4	21	24	11	6	8494	24320	82	222
(4 MPORTs)	5	4	3	2	5686	17877	272	216
	3	2	2	1	5913	14276	532	217

Table 6.3: Synthesis results for the considered variants of the IDCT HWPU

	Data Footprint: INPUT=676B, OUTPUT=4B						
	Ker	Operands		Regs	Area	Lat.	Freq
	Dim	mul	add	(1bit FF)	(slices)	(cycles)	(MHz)
Parallelism 1	7x7	2	1	112	158	54	220
(1 MPORT)	11x11	2	1	149	301	126	243
	13x13	2	1	153	372	175	247
Parallelism 2	7x7	4	2	208	373	30	260
(2 MPORTs)	11x11	4	2	219	384	66	276
	13x13	4	2	225	486	90	289
Parallelism 4	7x7	8	4	438	820	18	262
(4 MPORTs)	11x11	8	4	419	741	39	268
	13x13	8	4	417	787	48	273

Table 6.4: Synthesis results for the considered variants of the GAUS HWPU

and data bandwidth/speed. In the experiments, all HWPUs have 4 MPORTs. The input/output data footprint is also indicated. It is important to stress that the simulation infrastructure correctly models the behavior of MPORTs in case of a longer-latency memory operation due to data conflicts in the TCDM.

Figure 6.7 shows the results of the comparison of the above described acceleration strategies for the IDCT and GAUS kernels. The considered platform has a single core and a single accelerator. The Y-axis shows execution cycles, normalized to the sequential software implementation.

Three variants for *C-ACC* accelerators are studied, respectively considering data transfers from L1 (representative of AVS (20)), L2 (representative of Buffer Integrated Cache (50)) and L3 (representative of GPUs (5, 101)) memory. Each of these test cases has two further variants: a naive *copyin/execute/copyout* scheme and a pipelined implementation, using double buffering to overlap communication and computation. They are called respectively *C-ACC* and *C-ACC (DB)*.

Focusing on IDCT, the *ZC-HWPU* performs better than any other variants. Its performance is comparable to the *C-ACC (DB)* when considering transfers from a L1 memory, and allows a speedup of $\approx 3.12\times$ w.r.t *C-ACC*, and $\approx 2.12\times$ w.r.t *C-ACC (DB)* when considering transfers from L2. The speedups increase to $\approx 14.35\times$ and $\approx 13.35\times$ when considering transfers from L3. *ASIP* is roughly 12 times slower.

Focusing on GAUS, the plot refers to a single instance of the kernel, where each pixel of the image is processed based on the 13×13 neighboring elements (i.e. 169 input pixels, 1 output pixel). For the *C-ACC* approach, however, copy-in a 13×13 block are copied in only for the first pixel in a row, and for the successive pixels only copy-in a 13-pixel border are copied because we can reuse the remaining pixels copied with previous transfers. Due to the small amount of transferred data, when communicating through L1 and L2 memory the computation time dominates, and the double buffering scheme of *C-ACC (DB)* allows to hide all the communication cost, thus achieving equal performance to the *ZC-HWPU*. However, *ZC-HWPU* obtains a speedup of $\approx 1.2\times$ and $\approx 1.43\times$ against single-buffering *C-ACC* from L1 and L2 respectively. When data transfers take place through the L3 memory, *ZC-HWPU* is $\approx 2.56\times$ faster than *C-ACC (DB)* and $\approx 3.68\times$ faster than *C-ACC*. *ASIP* is roughly 4.52 times slower than *ZC-HWPU*.

6.4.3.2 Single HWPU Sharing

This section shows the effectiveness of the programming model by mixing parallelization with acceleration on a single *ZC-HWPU* shared among all cores in the system. The

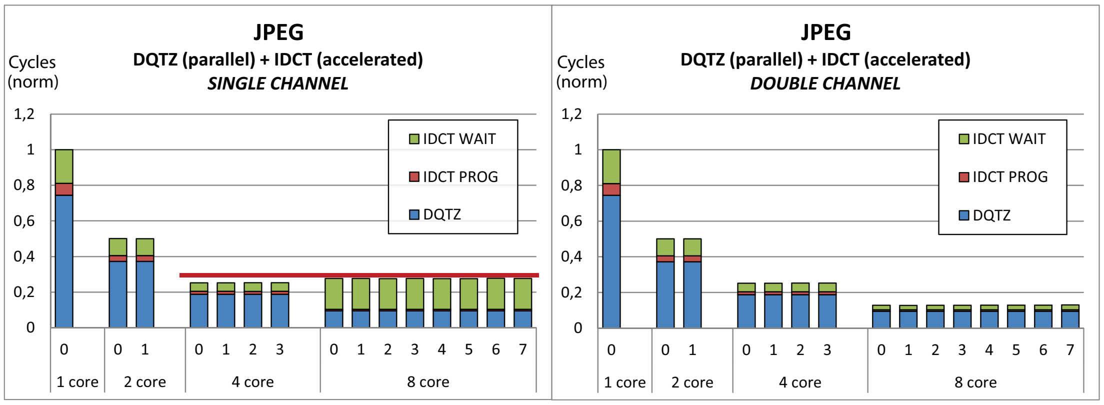


Figure 6.8: Effect of sharing a single HWPU among an increasing number of cores on JPEG

parallelization/acceleration strategy has been presented in Figures 6.5 and 6.6.

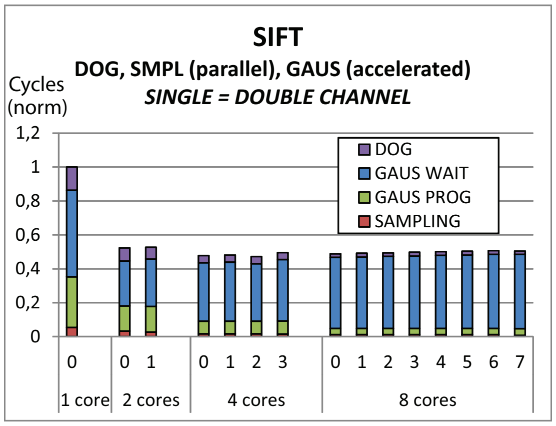


Figure 6.9: Single HWPU sharing on SIFT

JPEG was parallelized so that each macroblock is processed independently on a two-stage pipeline composed of DQTZ (executed on the target processor) and IDCT (offloaded on the *ZC-HWPU*). With this acceleration scheme several processors will be concurrently offloading computation onto a single HWPU, thus potentially incurring significant contention. Here we see how the system scales when an increasing number of cores shares the single accelerator, and how the double programming channel feature described in Section 6.3 possibly relieves the effect of heavy contention. The same experiment has been conducted for the SIFT benchmark as well. Application is partitioned onto three different parallel regions for the three main kernels: SMPL, GAUS, DOG. It is important to underline that with these scheme a barrier is implied at the end of each parallel region. Different from JPEG, where within a single loop body execution processors would asynchronously execute the DQTZ kernel and offload computation for the IDCT kernel, here the barrier implies that within the GAUS kernel processors only offload computation on the HWPU, and remain otherwise idle.

Results for this exploration are shown in Figures 6.8 and 6.9 respectively. For the JPEG it is possible to see that using HWPUs with a single programming channel the system scales well up to 4 cores. With 8 sharers the HWPU can not handle all offloading requests, and the effect of contention makes execution slower than with 4 cores. However, allowing processors to exploit a second programming channel to assign their jobs to the HWPU efficiently removes the source of contention. For the SIFT benchmark only the results for the HWPU with double programming channels are shown, because they are almost identical to the single channel case. Here, as explained above, within the GAUS kernel processors only spend their time in programming the HWPU, so when more than 2 processors are concurrently trying to offload their jobs on the HWPU most of their time is spent waiting for one of the two channels to become available. This prevents the system from scaling to a higher number of processors.

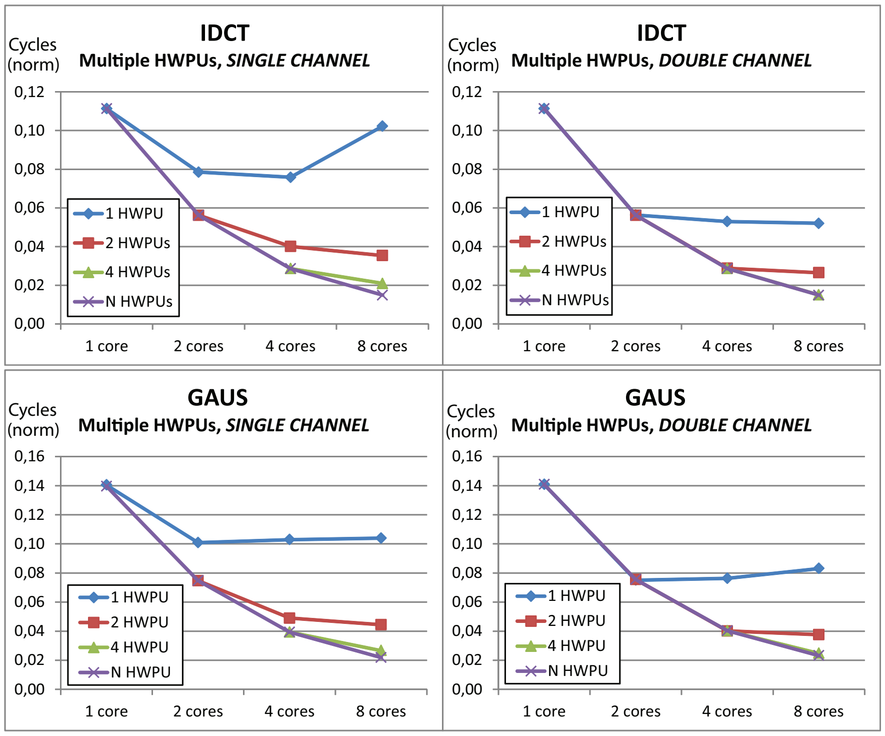


Figure 6.10: Multiple HWPUs, IDCT (top) and GAUS (bottom) kernel

6.4.3.3 Multiple HWPUs

Results in this section targets multiple identical HWPUs shared by processors (using the `num_hwpus` clause), focusing on the sole accelerated kernels. Results for this exploration are shown in Figures 6.10 for both IDCT (top) and GAUS (bottom). The Y-axis of these plots shows execution cycles normalized to the sequential software version. Different curves represent different system configurations where we leverage 1, 2, 4 or N (i.e., as many as processors) HWPUs. For both the kernels sharing a single HWPU scales only up to 2 processors, even using double channels. Increasing the number of HWPUs to two scales well up to 4 processors, and a perfect scaling is achieved with 4 HWPUs and double channels.

6.5 Bottom-up approach: integrating accelerators in tightly coupled clusters

In the previous section, a design flow was described to design from scratch a system with core and accelerators, implementing the *zero-copy* communication scheme using the tightly-coupled banks of on-cluster memory. Here, the diametrically opposite approach is shown: while the previous one targeted the design of a platform, that is, a *top-down* approach, here it is assumed that the platform already exists, that is, a *bottom-up* approach. From this perspective, integrating accelerators within a tightly-coupled cluster with *zero-copy* communication poses two main challenges: i) how to achieve tightly-coupled shared-memory communication for a large number of accelerators and ii) how to efficiently expose accelerators to the software stack, to simplify accelerator-based application development.

Several efforts in academia (8, 37) – as well as the approach already shown – and industry (65) are pushing for solutions to ease programmability on accelerator-based platforms, as witnessed by initiatives such as Khronos OpenVX (128). With it, the Khronos vision working group attempts at driving industry consensus to create a cross-platform API standard to enable hardware vendors to implement and optimize accelerated computer vision algorithms. At the time this document is being written, the Khronos vision working group has not converged towards a standard for OpenVX yet. However, we will see how this "philosophy" can be effectively embraced, by defining a set of basic functionalities for computer vision and image processing (the own "standard" functions), and providing program interfaces for their support. The architectural scalability problem is quite more complex, and it will be explained with an example.

6.5.1 Architectural scalability problem

In the considered architecture, typically *clusters* leverage a low-latency, high-bandwidth, crossbar-like interconnection to support tightly-coupling of processing units (see for instance the one in Figure 6.11(a)). However, the area/power cost for such an interconnection can only be afforded for a small number of computing elements, as its complexity becomes quickly too important. Figure 6.11(b) shows the area increment (μm^2) with the number of master ports for an interconnection system modeled after that proposed by Plurality for its HAL processors (109). Adding a high number of data ports for the accelerators i) increases the interconnect area and ii) generates longer critical paths, which eventually reduce the maximum achievable frequency target. In addition, typically accelerators increase efficiency by leveraging data parallelism to generate

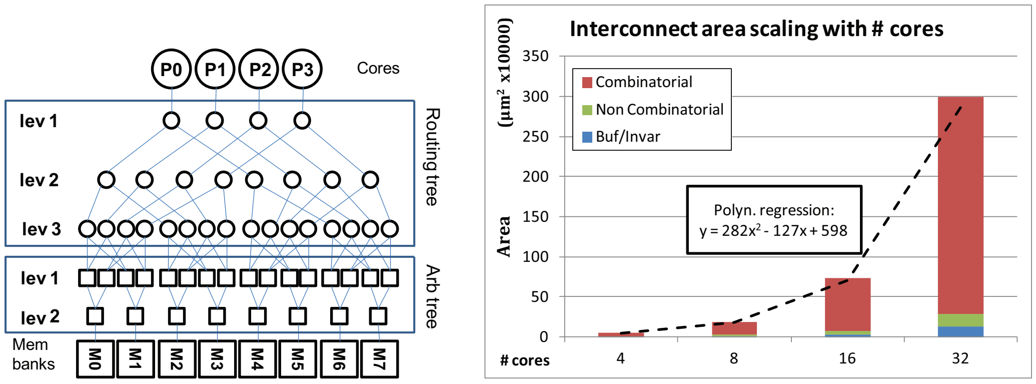


Figure 6.11: Scheme (a) and mathematical area model (b) of the Plurality intra-cluster interconnection (109, 111) as a function of the number of connected master ports

SIMD-like datapaths, which require more data ports, exacerbating this problem. To avoid the system becoming unmanageable, the interconnection exposes only a limited bandwidth (data ports), to be shared between cores and accelerators.

Here, 16 MPORTs are considered (see (109), (25)), and explore two different configurations of the heterogeneous cluster. One where 1/4 of the MPORTs is used for accelerators, and one where 1/2 of the MPORTs is used.

6.5.2 Data Pump scheme

To address the problem of plugging several accelerators to the interconnection system without increasing its complexity too much, the **Data Pump** is introduced: a module that multiplexes data access requests to/from the accelerators on the available master ports (from now on, called **DP-MPORTs**) of the main cluster interconnect. Figure 6.12 shows the generic cluster with M cores, N accelerators, and the Data Pump. The block diagram of the Data Pump structure is shown in Figure 6.13. The Data Pump stores a table containing the features about the accelerators (class, number of accelerators per class, number of master port, data set size). It is split in two parts: one *Controller*, which handles offload requests to the HWPUs, and a *Master*, which handles data requests from HWPUs. They will now be described in details.

The *Controller* exposes a memory-mapped slave port (**DP-SPORT**) on the interconnect, to support offloading sequences by the cores to the HWPUs through their SPORTs. It provides different *programming modes*:

1. The **original programming mode (OPM)**. In this case, a core directly references a specific accelerator instance through its unique ID, and the Data Pump

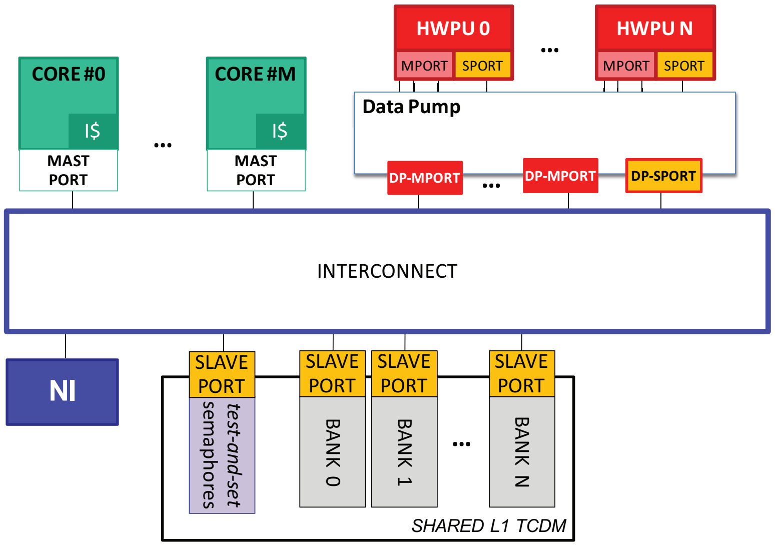


Figure 6.12: Heterogeneous shared memory cluster template with Data Pump.

transparently propagates the request to the target module. This corresponds to a "standard" programming sequence, as shown in Section 6.3 and in Dehyadegari's work (39), where accelerators are directly exposed to the interconnection system and the software layer.

2. With the **class programming mode (CPM)**, the concept of *accelerator classes* is introduced, that match one or more HWPUs in the system. In this model an offload sequence specifies an *accelerator class* to offload to, and the Data Pump will dispatch the request to one of the HWPUs implementing it, hiding hardware scheduling details to the software level. Software-level support for this programming mode, as well as *accelerator classes* are described in details in Section 6.5.3. Internally, the offload request is stored in a class-specific queue. A unique ID is returned to the calling program, and can be used for instance to implement synchronization. The *Controller* is in charge of fetching the requests from the queues, and to dispatch them to the target HWPU. To do so, it owns a table which binds every accelerator to a specific class. The choice of which HWPU will execute a given *task* is implementation-specific.

3. A quite common pattern for image processing applications is the one of executing consequently the same algorithm on independent (i.e., non-overlapping) data sets, such as image blocks. To support this, a **block programming mode (BPM)** is provided, where in addition to the class and in/out data addresses, a number

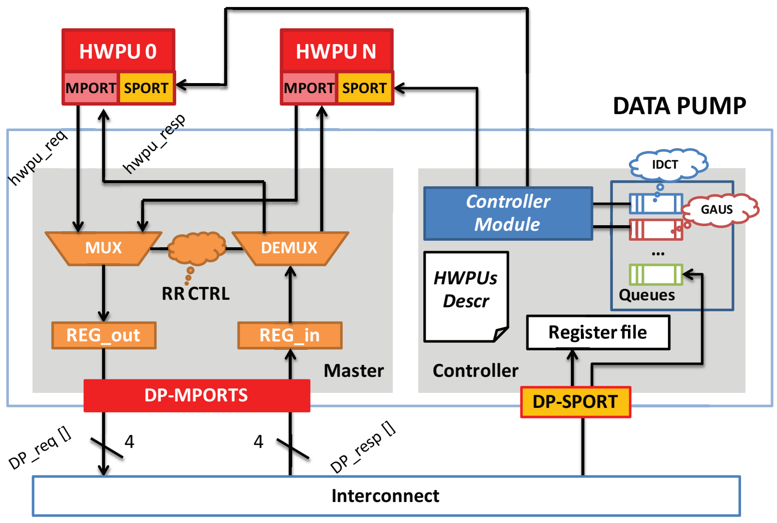


Figure 6.13: Scheme of the Data Pump

n_{iter} of consecutive executions can be specified, which matches e.g., the number of image blocks. A *stride* is also specified for each input and output data, to compute data address offsets in subsequent HWPU invocations. A few additional logics (e.g., a register file) are needed inside the *Controller* submodule to support it.

The *Master* submodule is in charge of handling data requests from several HWPU MPORTs, and to redirect them towards the memory system.

For that, it implements a round robin arbiter which pilots a MUX and a DEMUX for arbitrating between the different requests (see Figure 6.13). The round robin scheme works at the granularity of the single port. Each DP-MPORT holds registers to store the in/out data, and its address for the current transaction. Thus, a two-cycle delay is added to traverse the Data Pump logic. To avoid this delay, a fully combinational *Master* block could be implemented. However, the interconnect itself being fully combinational may lead to too long critical paths to meet the target design frequency: this research path is still unexplored.

Most of the HLS tools perform loop unrolling to increase the I/O parallelism of the accelerator. As a consequence, the tool typically schedules several data accesses to happen in the same clock cycle of the HWPU FSM, through the different MPORTs. In such a template, if a single data access is delayed (for instance, for a conflict on SPM banks or on the DP-MPORTs allocation), the full set of data requests for that FSM state is stalled. During this time the DP-MPORTS are not reallocated. This issue can be tackled in several ways (implementing “smarter” arbitration schemes, or data prefetching, or dimming the clock frequency of HW blocks to hide memory latency), and it is left as a future research.

6.5.3 Class-based programming extensions

The `#pragma omp accelerate` directive (shown in section 6.4.1) is extended to support Class Programming Mode, and more in details a `class` clause is introduced.

```
#pragma omp accelerate class ("<string>", var) [clause[[,]clause]...]
    structured-block
```

The `class` clause is mandatory, and specifies which *kernel* class the code region identifies, as defined by the standard. This eventually matches one of the accelerator classes shown in Table 6.5, and detailed later in this section. The structured block enclosed within an `accelerate` directive implements the SW version of the specified kernel functionality. The compiler will eventually resort to the SW implementation when the corresponding HW implementation is not available (or all the HWPUs are busy for a long time). As previously explained, other *clause*s can be one of the following, and are optional:
`private (`*list*`)`
`firstprivate (`*list*`)`
`lastprivate (`*list*`)`
`shared (`*list*`)`
`nowait`
The `num_hwpus` clause is not considered in this scenario.

The following code shows a basic usage of the directive.

```
int handle, data[1024];

#pragma omp accelerate class ("IDCT", handle) shared (data)
{
  idct_code (data, data);
}
```

The code is transformed so as to perform lookup of the available HWPUs in the platform, and a call into an appropriate library function is inserted to offload computation to the target HWPU. The offloading routine returns an unique ID, which can be used later as a handle to perform synchronization. Pointers to non-private data are created and passed as parameters to the offloading routine. In this example, the array

Algorithm *class*	Description	Library function	Params
CSC	Color Space Conversion RGB → HVS	__builtin_omp_library_CSC	(in) img (out) img
Detect_VJ	Viola-Jones Detection Single cascade detection	__builtin_omp_library_DetectVJ	(in) II img, II^2 img (out) result (app.specific)
Detect_HoG	HoG Detection	__builtin_omp_library_DetectVJ	(in) img (out) result (app.specific)
Gaus	Gaussian Blur 2 X (1D-convolution)	__builtin_omp_library_Gaus	(in) img (in) ker (out) img
Gradient	Gradient	__builtin_omp_library_Gradient	(in) img (out) img
IDCT	Inverse DCT	__builtin_omp_library_IDCT	(in) img (out) img
IntegralImage	Integral Image	__builtin_omp_library_IntegralImage	(in) img (out) II img

Table 6.5: Accelerator classes

data is specified as in/out data using the (default) `shared` clause. The transformed code looks like the following:

```
int handle, data[1024];

if (omp_query_HWPU("IDCT"))
{
  /* Array data was declared SHARED => I/O */
  handle = __builtin_omp_library_IDCT (data, data);
  /* Synch */
  omp_wait_HWPU (handle);
}
else
{ /* Lookup failed: run SW version */
  idct_code(data, data);
}
```

The `omp_query_HWPU` primitive inspects the platform to check if there are any HWPUs matching the specified *class* (in this case, "IDCT"). This is done by inspecting dedicated read-only registers in the Data Pump that store the configuration of the platform (initialized at boot time). Note that the allowed *class* names are pre-defined by the standard: this will be discussed later on, when the `class` clause is explained in details. If the lookup succeeds, then an appropriate stub is invoked to perform offloading. Input and Output data are inferred by the `firstprivate`, `lastprivate` and `shared` clauses and their addresses are passed as parameters to the offloading function. If no HWPU is present in the platform, the software version of the algorithm (extracted by the original code) is executed. If no `nowait` clause is specified, the call is synchronous, that is, the code running on host waits for the HWPU to end the execution. The runtime function `omp_wait_HWPU` supports ID-based synchronization with HWPUs, and in this example it is automatically inserted by the compiler.

6.5.3.1 Naming conventions and `class` clause

The kernel class specified with the `class` clause must match one entry of a set of standard-defined functionalities. Specific implementations of the runtime must provide a stub for each of them. The naming convention for the stubs is the following:

```
__builtin_omp_library_< CLASS_NAME > (..)
```

To ensure correct compilation and linking of the code, in case the platform does not provide a HW implementation for a given kernel, the stub will be empty, and the `omp_query_HWPU` will return *false* (it must anyway be defined, to make the code linking process possible). As a use-case embodiment of the proposed approach, the sample set of accelerated functions is proposed (shown in Table 6.5), along with the corresponding library function implementations. They target the domain of image processing and computer vision applications.

6.5.3.2 Synchronization

By default, upon encountering an `accelerate` region the processor is stalled until the offloaded task returns, unless the `nowait` clause is specified. In this case the offloading call is asynchronous and it becomes duty of the programmer to enforce synchronization where appropriate, using the directive:

```
#pragma omp accwait (var)
```

Note that the `var` parameter is the one specified in the `class` clause of the `accelerate` directive. So, an example of code performing asynchronous offloading might look like:

```
int handle = -1, /* Holds the ID of the offloaded HW task */
    data[1024];

#pragma omp accelerate class ("IDCT", handle) shared (data) nowait
{
  idct_code (data, data);
}

// more (asynch) code
// [...]

if (handle >= 0) /* If executing on HW need to sync here */
  #pragma omp accwait (handle)
```

The compiler transforms the code in:

```
int handle, data[1024];
if (omp_query_HWPU ("IDCT"))
{
  handle = __builtin_omp_library_IDCT (data, data);
  /* NO SYNCH HERE */
}
else
{ /* lookup failed: run SW version */
  ...
}

// more (asynch) code
// [...]

/* (manual) synch */
if (handle >= 0) /* If executing on HW need to sync here */
  omp_wait_HWPU (handle);
```

So, in case the offload succeeds, the host performs its own computation in parallel, until the synchronization point is reached. In case the offload fails (e.g., no HW support present in the platform) the synchronization point is simply skipped.

6.5.3.3 Runtime implementation

It is duty of the specific library function implementation to offload the *hardware task* to the Data Pump HWPUs controller. Low-level primitives are provided to support this by accessing Data Pump registers and *class* queues. The following snippet of code shows their basic usage inside the implementation of the *IDCT* standard library function.

```
int
__builtin_omp_library_IDCT (int mdata_ptr[])
{
  int handle;
  while (DP_queue_full (IDCT_QUEUE_ID))
    ; /* If queue is full, wait */

  /* Acquire the Data Pump */
  DP_lock ();

  task_ID = DP_reserve_queue (IDCT_QUEUE_ID);
  /* Now we exclusively own the queue for
    "IDCT" HW tasks */
  DP_set_inaddr (IDCT_QUEUE_ID, mdata_ptr[0]);
  DP_set_outaddr (IDCT_QUEUE_ID, mdata_ptr[1]);

  /* End of the programming sequence.
     Release the Data Pump so that other
     tasks can be enqueued */
  DP_unlock ();

  return handle;
}
```

It is important to remark here that no data movements are needed here; since communication is *zero-copy*, HWPUs just need to be aware of the **location** (addresses) of data, that they will directly read from the shared memory system.

6.5.4 Experimental results

The proposed heterogeneous shared memory cluster was prototyped using the cycle-accurate Virtual Platform VirtualSoC (26), with main architectural parameters as summarizes in Table 6.6. As explained, two architectural variants are considered, namely one with 12 cores and 4 DP-MPORTs, and one with 8 cores and 8 DP-MPORTs, for a total of 16 MPORTs on the interconnection. With this setup, two applications from the image processing and computer vision domain were run, namely a JPEG decoder and the Viola-Jones algorithm for face detection (137). The purposes is to validate the programming model and characterizing the architecture in terms of performance and energy efficiency. SystemC models for the Data Pump and all the accelerators were developed and included in the virtual platform, as well as RTL models which was synthesized with Synopsys Design Compiler (119) to gather area and energy estimates.

To increase the locality of data to HWPUs, they must reside in the on-cluster SPM. Big images are split in slices, which were manually moved to/from the off-chip L3

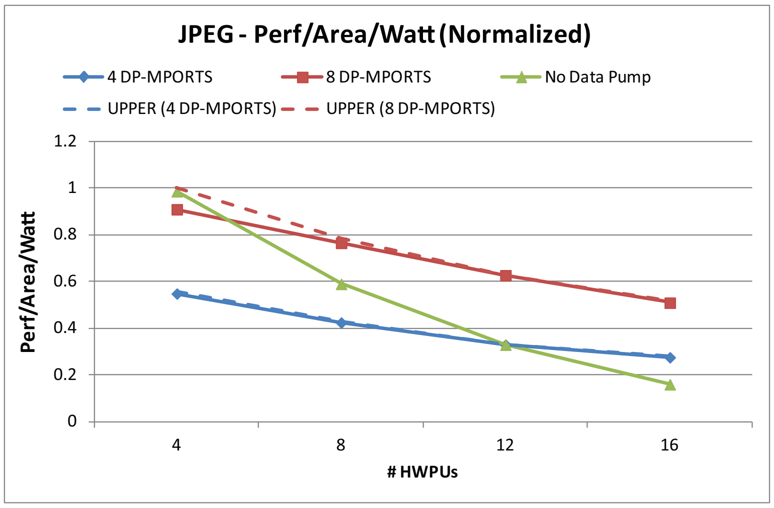


Figure 6.14: Normalized Performance/Area/Watt for accelerated JPEG decoder

ARM v6 cores	*c1*:12 *c2*:8	DP-MPORTs	*c1*:4 *c2*:8
L1 SPM size	256 KB	# L1 SPM banks	32 (K=2)
L3 size	256 MB	L3 latency	$\geq$ 59 cycles
$I\$_i$ size	1 KB	$I\$_i$ line	4 words
t_{hit}	= 1 cycle	t_{miss}	$\geq$ 59 cycles

Table 6.6: Architectural parameters

memory with DMA transfers. Communication latency is hidden with DMA double buffering: this is a quite common decomposition style for image processing systems (132, 145).

6.5.5 JPEG decoder

Figure 6.5 shows the structure of a JPEG decoder. It is composed of four main kernels: Huffman AC and DC coefficient computation, luminance dequantization (LD) and inverse discrete cosine transform (IDCT). The focus is on LD and IDCT kernels, which were synthesizing HWPUs with an unroll factor of 4 (i.e., with 4 MPORTs), a typical optimization to exploit I/O parallelism in accelerator. As a HLS tool we used GAUT. Two implementations were provided for each of the LD and IDCT kernels. A first implementation called *naive*, whose datapath was designed "by-hand", which fetches the full data set (made of, by standard, 8x8 image blocks) before performing computation, and resulting image is stored afterwards, thus following a a LOAD-EXEC-STORE pat-

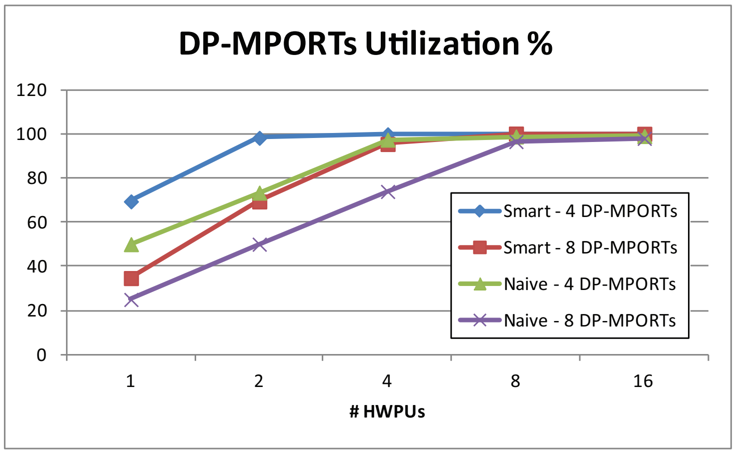


Figure 6.15: DP-MPORTs Utilization for JPEG HWPUs

tern. The second implementation is called *smart*, because it is capable of overlapping execution and data accesses, and it was designed it using GAUT (133).

The target architecture is compared against the baseline heterogeneous shared memory cluster (i.e., a system with no Data Pump). It is important to stress that while for the experiments the baseline architecture was instantiated with up to 32× 4-port HWPUs (128 ports) directly to the interconnect, there is a maximum physical limit of 32 ports, beyond which it is not possible to synthesize the whole cluster at the target frequency (500 MHz).

Results plot the Performance/Area/Watt ($\frac{1}{cycles*\mu m^2*mW}$) of the accelerated application, considering a cluster without Data Pump (No DP – the baseline architecture), and two clusters with Data Pumps having 4 and 8 DP-MPORTs, respectively. Performance was measured on the *smart* implementation of HWPUs. To do so, the cumulative area and power were measured for each configuration. Figure 6.14 shows the results, normalized to the maximum value. When only 4 (or less) accelerators are considered, the Data Pump does not show any advantage, because the area/energy increase for the baseline cluster is not significant. It must be noted anyhow that the 8 DP-MPORT design delivers comparable results to the baseline. When the number of HWPUs is increased to 8, the Data Pump provides much better results than the baseline. The dashed lines are obtained with a mathematical model which represents the maximum achievable performance for an ideal Data Pump which is data-dominated (i.e., at every cycle all the ports are servicing a new request). This is computed taking into account the data request rate and the data set size, and represents a upper bound on the achievable results. Dotted lines in charts (actually, the line for 4 DP-MPORTs overlaps the results line), Results show that the Data Pump-based systems are delivering results very close to their full potential. This stems from the fact that HWPUs for JPEG algorithm have significant memory boundedness (i.e., more than 50%) , as shown in Figure 6.15, which shows the percentage of cycles the DP-MPORTs are busy, normalized to the overall HWPU execution time. JPEG application have a regular execution pattern,

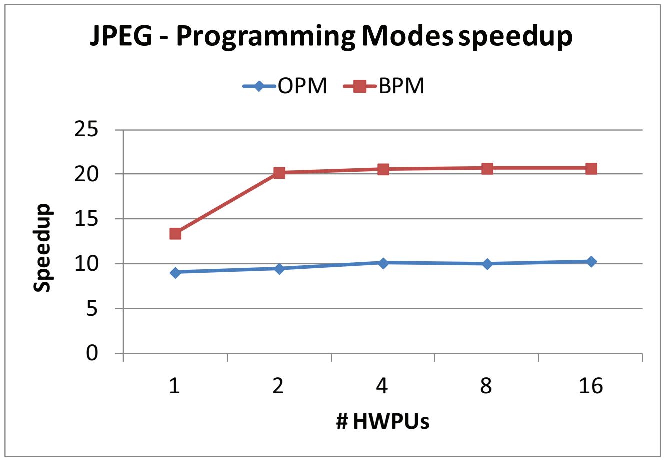


Figure 6.16: Comparison of Original and Block Programming model (JPEG application)

whose *kernels* consecutively process non-overlapping image blocks. Hence, Block Programming Mode can be adopted. Figure 6.16 compares speedups obtained using the Original Programming Mode (OPM) and the Block Programming Mode (BPM). It is evident the improvement achieved by the latter. Unfortunately, such a programming model cannot be adopted with the OpenMP frontend, without heavily overloading its semantics. OpenACC (105), on the other hand, seems a better candidate, and this exploration is left as a future work.

6.5.6 Face Detection

The Viola-Jones algorithm (137) is widely adopted for practical face detection systems. It is based on To do so, it scans an image, applying a set of so-called *cascades* to identify predefined patterns (for instance, eyes). Figure 6.17 shows the block diagram of the application. It is organized as a set of nested loops iterating over different image scales, image stripes, (integral) image windows, cascades (i.e., trained sets of Haar features) and different orientations (rotations) of the same cascade.

After profiling the application, a good candidate for acceleration was identified in the kernel which applies the cascade on the target image window, searching for matching features. Table 6.7 shows the results of the profiling. Note that this core function can

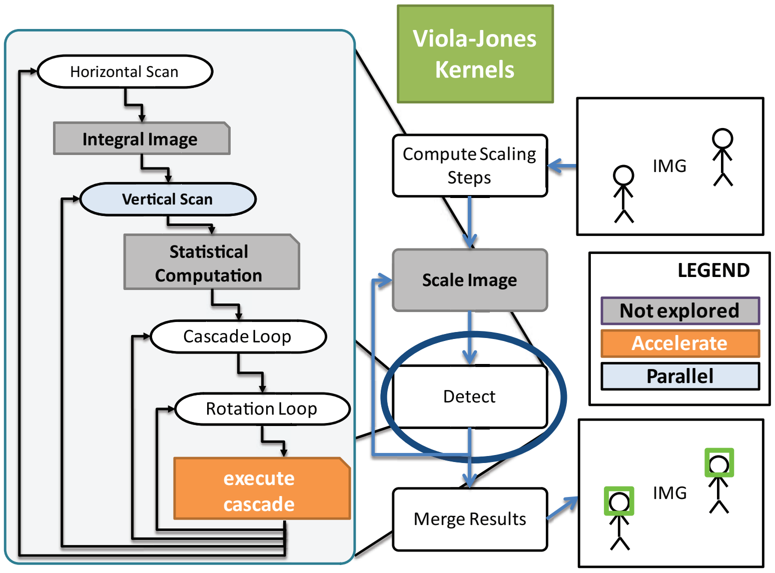


Figure 6.17: Block scheme for Viola-Jones face detector

Kernel	Tot cycles	%mem
`execute_cascade`	21697626	11%
`compute_integral`	16480006	18%
`scale`	3223930	<1%
`compute_stats`	1085078	18%

Table 6.7: Viola Jones main *kernels* sorted by execution time

be aborted for increasing performance in case a patch is recognized as "non-interesting" with high probability. Thus, the exact execution time of the *kernel*, as well as its exact memory boundedness, cannot be predicted in advance.

Figure 6.18 shows performance/area/watt results of the experiments with the face detection, comparing the baseline shared memory cluster with the Data Pump-based ones. Similar results to JPEG hold also in this case. When the number of accelerators in the system increases, the Data Pump provides up to 20% better numbers than the baseline cluster. Note that the upper bound to achievable results in this benchmark is not measured, since it is not possible to accurately model the accelerator behavior in terms of memory accesses due to data-dependent control flow. Results for this application are slightly better from the JPEG ones (Figure 6.14) due to the different memory boundedness.

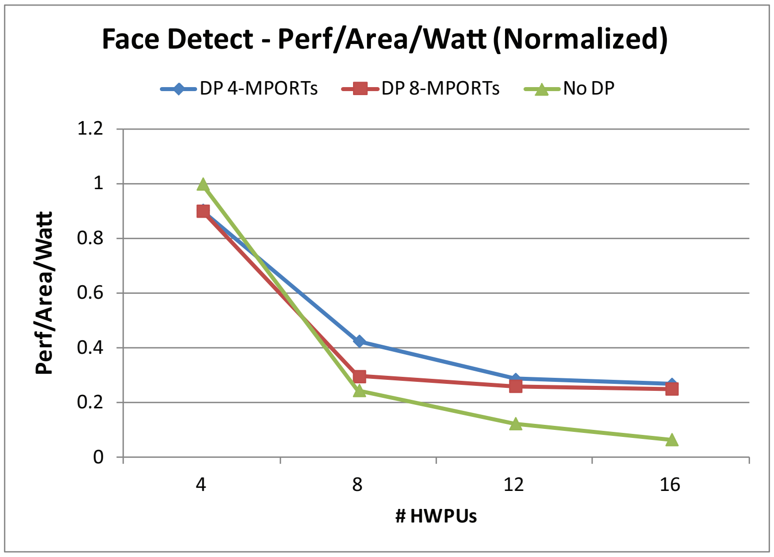


Figure 6.18: Normalized Performance/Area/Watt for accelerated Viola-Jones face detector

6.6 Conclusions

Embedded platforms are increasingly embracing the heterogeneous paradigm, to take advantage of hardware accelerators. In this chapter, an architecture was proposed where hardware accelerators are tightly-coupled to cores inside many-core clusters, and *zero-copy* communication was implemented by sharing the memory and the interconnection system. Here, OpenMP was adopted for designing the platform, and for increasing its *programmability*. To do so, a custom `pragma omp accelerate` was introduced to annotate portions of code suitable for acceleration in hardware. As a first contribution, a *top-down* approach was considered, and a design flow was proposed that starting from an application, automatically generates an heterogeneous platform where accelerators are created out of annotated portions of code using an HLS tool (133). A second part considered the issues of programmability and architectural scalability of the target cluster template, and tackled them by introducing the Data Pump module. Under this *bottom-up* light, the Data Pump acts as a “virtualization” layer for accelerators, and each of them is mapped to a more generic *accelerator class*. Using this abstraction, a small set of standard APIs was provided for developing modular code in an efficient manner. The proposed OpenMP frontend leverages on this API layer to automatically detect and exploit accelerators that are already existing in the platform. If the architecture do not provide accelerating opportunities, code dynamically self-adapts so to resort to a pure-SW version of the application, thus it is portable *as-is* also on a different (homogeneous) platforms. An extensive set of experiment validated the proposed techniques on a cycle-accurate virtual platform (26), and demonstrated that the use of OpenMP enables a productive mix of parallelization and hardware acceleration.

7

Conclusions

Many-core architectures have been adopted for embedded systems designs, to keep performance scaling going on, while meeting the increasingly stringent power budgets requested by the market. These platforms have a tremendous potential in terms of parallelism and energy efficiency, but the task of extracting their peak performance is more and more demanded at the software layer, and at programmers' skills. Shared-memory communication is an effective mechanism for increasing performance, but when applying to modern platforms, which replace data caches with software-managed scratchpad memories, it harnesses programmability.

Several programming models exist to support the development of parallel code under a shared-memory assumption, but efficiently exploiting them on modern architectures is not easy, because the necessary runtime support introduces an overhead that cannot be tolerated in embedded systems, which are resource-constrained. This dissertation explored the applicability of the shared-memory paradigm on many-core embedded systems, from a software perspective, and more in details, when adopting the OpenMP programming model.

A first part of the thesis analyzed the costs of basic services (i.e., synchronization and workload distribution) for supporting an expressive parallel programming model such as OpenMP (106) on these architectures. They were characterized, and techniques were proposed aimed at improving them. Then, a custom runtime (`libgomp` (52)) was proposed, which efficiently supports multi-level and irregular parallelism on the target architectures, in a scenario where tasks are fine-grained, and traditional solutions were shown not to be effective anymore.

The second part of the thesis explored the applicability of shared-memory communication to heterogeneous platforms, and proposed a template for a scalable many-core cluster featuring tightly-coupled hardware accelerators. A design methodology for such a platform was showed, together with a complete software stack for efficiently exploiting the accelerators from the OpenMP frontend, using a small set of proposed extensions (*pragmas*). Architectural scalability issues were also tackled, by introducing the Data Pump module, which also provides lightweight low-level support for efficiently programming the accelerators.

7.1 Future research directions

In chapter 5, we showed a novel design for the OpenMP runtime support, targeting a shared-memory tigthly coupled cluster. No NUMA effects due to the multi-cluster environment were considered, and taking them in account is the natural short-range research path. For instance, the *tree barrier* proposed in chapter 3 can be effectively adopted in a multi-cluster environment, as well as the meta-data replication schemes. Similarly, the approach presented in chapter 4 can be adopted, for instance, inside the OpenMP task dispatcher (chapter 5), so that knowledge on data partitioning schemes can be used to maximize the locality of data to the processing clusters. This issues are still unexplored.

Similar considerations hold for chapter 6, which targets a single heterogeneous cluster. When moving to multi-cluster, what is more interesting is the problem of *where* actually placing the accelerators, while designing the platform. Is it better to have a single *hardware* cluster with all accelerators? Or to have *symmetric* identical clusters? Or again, grouping accelerator of the same kind (i.e., GAUS, IDCT, CSC) in the same cluster? This questions are still without answer.

A more longer-term goal refers to programmability, and consists in either studying the applicability to many-core (heterogenous) clusters of the most recently proposed languages (such as OpenACC (105) of the latest – device-aware – OpenMP 4.0 (107)), or keeping following the standardization trend, similarly to what was shown in chapter 6, for instance as OpenVX (128) becomes officially a standard.

Finally, considering heterogeneous architectures, a current “hot” topic are neural-network and bio-inspired computing (48, 84): the tasks of efficiently coupling many-cores and neural accelerators and to effectively expose them at the application layer are quite challenging and exciting. Looking from another perspective, many-core architectures can also be used to *support* neural network computing, by simulating neurons on many-cores, and communication happens through the shared-memory.

8

Publications

2010

Paolo Burgio, Martino Ruggiero, Francesco Esposito, Mauro Marinoni, Giorgio C. Buttazzo, Luca Benini: *Adaptive TDMA bus allocation and elastic scheduling: A unified approach for enhancing robustness in multi-core RT systems.* ICCD 2010: 187-194.

Andrea Marongiu, Paolo Burgio, Luca Benini: *Evaluating OpenMP Support Costs on MPSoCs.* DSD 2010: 191-198.

Andrea Marongiu, Paolo Burgio, Luca Benini: *Vertical stealing: robust, locality-aware do-all workload distribution for 3D MPSoCs.* CASES 2010: 207-216.

2011

Andrea Marongiu, Paolo Burgio, Luca Benini: *Supporting OpenMP on a multi-cluster embedded MPSoC.* Microprocessors and Microsystems - Embedded Hardware Design 35(8): 668-682 (2011).

Alessio Franceschelli, Paolo Burgio, Giuseppe Tagliavini, Andrea Marongiu, Martino Ruggiero, Michele Lombardi, Alessio Bonfietti, Michela Milano, Luca Benini: *MPOpt-Cell: a high-performance data-flow programming environment for the CELL BE processor.* Conf. Computing Frontiers 2011: 11.

Jakob Rosen, Carl-Fredrik Neikter, Petru Eles, Zebo Peng, Paolo Burgio, Luca

Benini: *Bus Access Design for Combined Worst and Average Case Execution Time Optimization of Predictable Real-Time Applications on Multiprocessor Systems-on-Chip.* IEEE Real-Time and Embedded Technology and Applications Symposium 2011: 291-301.

2012

Andrea Marongiu, Paolo Burgio, Luca Benini: *Fast and lightweight support for nested parallelism on cluster-based embedded many-cores.* DATE 2012: 105-110.

Paolo Burgio, Andrea Marongiu, Dominique Heller, Cyrille Chavet, Philippe Coussy, Luca Benini: *OpenMP-based Synergistic Parallelization and HW Acceleration for On-Chip Shared-Memory Clusters.* DSD 2012: 751-758.

2013

Paolo Burgio, Giuseppe Tagliavini, Andrea Marongiu, Luca Benini: *Enabling fine-grained OpenMP tasking on tightly-coupled shared memory clusters.* DATE 2013: 1504-1509.

Abbas Rahimi, Andrea Marongiu, Paolo Burgio, Rajesh K. Gupta, Luca Benini: *Variation-tolerant OpenMP tasking on tightly-coupled processor clusters.* DATE 2013: 541-546.

Paolo Burgio, Andrea Marongiu, Robin Danilo, Philippe Coussy, Luca Benini: *Architecture and Programming Model Support for Efficient Heterogeneous Computing on Tigthly-Coupled Shared-Memory Clusters.* DASIP 2013, 22-29.

2014 – to appear

Paolo Burgio, Giuseppe Tagliavini, Francesco Conti, Andrea Marongiu, Luca Benini: *Tightly-Coupled Hardware Support to Dynamic Parallelism Acceleration in Embedded Shared Memory Clusters.* DATE 2014.

Paolo Burgio, Andrea Marongiu, Robin Danilo, Philippe Coussy, Luca Benini: *A tightly-coupled Hardware Controller to improve scalability and programmability of*

shared-memory heterogeneous clusters. DATE 2014.

References

[1] Adapteva, Inc. **Epiphany-IV 64-core 28nm Microprocessor**. [Online] http://www.adapteva.com/products/\silicon-devices/e64g401/, 2013. 1, 7

[2] S.N. Agathos, P.E. Hadjidoukas, and V.V. Dimakopoulos. **Design and Implementation of OpenMP Tasks in the OMPi Compiler**. In *Informatics (PCI), 2011 15th Panhellenic Conference on*, pages 265–269, 2011. 14, 86, 87, 109, 113, 120, 122, 123

[3] Kunal Agrawal, Charles E. Leiserson, Yuxiong He, and Wen Jing Hsu. **Adaptive work-stealing with parallelism feedback**. *ACM Trans. Comput. Syst.*, **26**(3):7:1–7:32, September 2008. [Online] http://doi.acm.org/10.1145/1394441.1394443. 60

[4] Sameer Alawnah and Assim Sagahyroon. **Modeling smartphones power**. In *EUROCON, 2013 IEEE*, pages 369–374, 2013. 1, 3

[5] AMD. **The AMD Fusion Family of APUs**. [Online] http://sites.amd.com/us/fusion/apu/Pages/fusion.aspx. 1, 6, 86, 125, 142

[6] R Anigundi, Hongbin Sun, Jian-Qiang Lu, K. Rose, and Tong Zhang. **Architecture design exploration of three-dimensional (3D) integrated DRAM**. In *Quality of Electronic Design, 2009. ISQED 2009. Quality Electronic Design*, pages 86–90, 2009. 58

[7] Inc. Apple. **The Grand Central Dispatch** . [Online] http://images.apple.com/macosx/technology/docs/GrandCentral_TB_brief_20090608.pdf. 3, 14, 86, 121

[8] C&#x00e9;dric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-Andr&#x00e9; Wacrenier. **StarPU: a unified platform for task scheduling on heterogeneous multicore architectures**. *Concurr. Comput. : Pract. Exper.*, **23**(2):187–198, February 2011. 129, 146

[9] Eduard Ayguad, RosaM. Badia, Pieter Bellens, Daniel Cabrera, Alejandro Duran, Roger Ferrer, Marc Gonzlez, Francisco Igual, Daniel Jimnez-Gonzlez, Jess Labarta, Luis Martinell, Xavier Martorell, Rafael Mayo, JosepM. Prez, Judit Planas, and EnriqueS. Quintana-Ort. **Extending OpenMP to Survive the Heterogeneous Multi-Core Era**. *International Journal of Parallel Programming*, **38**(5-6):440–459, 2010. [Online] http://dx.doi.org/10.1007/s10766-010-0135-4. 129

[10] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Yuan Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and Guansong Zhang. **The Design of OpenMP Tasks**. *Parallel and Distributed Systems, IEEE Transactions on*, **20**(3):404–418, 2009. 19, 87

[11] Eduard Ayguade, Xavier Martorell, Jesus Labarta, Marc Gonzalez, and Nacho Navarro. **Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study**. In *Proceedings of the 1999 International Conference on Parallel Processing*, ICPP '99, pages 172–, Washington, DC, USA, 1999. IEEE Computer Society. [Online] http://dl.acm.org/citation.cfm?id=850940.852871. 85, 86

[12] L.A.D. Bathen and N.D. Dutt. **Software Controlled Memories for Scalable Many-Core Architectures**. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on*, pages 1–10, 2012. 1, 2

[13] T.F. Beatty, E.E. Aubanel, and K.B. Kent. **An OpenMP-based circuit design tool: Customizable bit-width**. In *Communications, Computers and Signal Processing, 2009. PacRim 2009. IEEE Pacific Rim Conference on*, pages 17–22, 2009. 130

[14] L. Benini and G. De Micheli. **Networks on chips: a new SoC paradigm**. *Computer*, **35**(1):70–78, 2002. 7

[15] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. **P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator**. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 983–987, 2012. 1, 2, 7, 9, 25, 88, 106

[16] D. Bertozzi and L. Benini. **Xpipes: a network-on-chip architecture for gigascale systems-on-chip**. *Circuits and Systems Magazine, IEEE*, **4**(2):18–31, 2004. 30

[17] Himanshu Bhatnagar. *Advanced ASIC Chip Synthesis: Using Synopsys' Design Compiler and PrimeTime*. Kluwer Academic Publishers, Norwell, MA, USA, 1999. 3

[18] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C.A. Nelson, and C.D. Offner. **Extending OpenMP For NUMA Machines**. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 48–48, 2000. 28, 45

[19] John Bircsak, Peter Craig, RaeLyn Crowell, Zarka Cvetanovic, Jonathan Harris, C. Alexander Nelson, and Carl D. Offner. **Extending OpenMP for NUMA machines**. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society. [Online] http://dl.acm.org/citation.cfm?id=370049.370455. 60, 62

[20] P. Biswas, N.D. Dutt, L. Pozzi, and P. Ienne. **Introduction of Architecturally Visible Storage in Instruction Set Extensions**. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, **26**(3):435–446, 2007. 1, 128, 141, 142

[21] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, Lei Jiang, G.H. Loh, D. McCauley, P. Morrow, D.W. Nelson, D. Pantuso, P. Reed, J. Rupley, Sadasivan Shankar, J. Shen, and C. Webb. **Die Stacking (3D) Microarchitecture**. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 469–479, 2006. 60

[22] G. Blake, R.G. Dreslinski, and T. Mudge. **A survey of multicore processors**. *Signal Processing Magazine, IEEE*, **26**(6):26–37, 2009. 26

[23] Geoffrey Blake, Ronald G. Dreslinski, Trevor Mudge, and Krisztián Flautner. **Evolution of thread-level parallelism in desktop applications**. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 302–313, New York, NY, USA, 2010. ACM. [Online] http://doi.acm.org/10.1145/1815961.1816000. 2

[24] Shekhar Borkar and Andrew A. Chien. **The future of microprocessors**. *Commun. ACM*, **54**(5):67–77, May 2011. [Online] http://doi.acm.org/10.1145/1941487.1941507. 1

[25] D. Bortolotti, F. Paterna, C. Pinto, A. Marongiu, M. Ruggiero, and L. Benini. **Exploring instruction caching strategies for tightly-coupled shared-memory clusters**. In *System on Chip (SoC), 2011 International Symposium on*, pages 34–41, 2011. 88, 106, 147

[26] Daniele Bortolotti, Christian Pinto, Andrea Marongiu, Martino Ruggiero, and Luca Benini. **VirtualSoC: a Full-System Simulation Environment for Massively Parallel Heterogeneous System-on-Chip**. In *013 IEEE 27th International Symposium on Parallel & Distributed Processing Workshops and PhD Forum*, pages 2182–2187. IEEE, May 2013. 75, 106, 121, 137, 138, 154, 160

[27] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. **State-of-the-art in heterogeneous computing**. *Sci. Program.*, **18**(1):1–33, January 2010. [Online] http://dl.acm.org/citation.cfm?id=1804799.1804800. 3

[28] P. Burgio, A. Marongiu, D. Heller, C. Chavet, P. Coussy, and L. Benini. **OpenMP-based Synergistic Parallelization and HW Acceleration for On-Chip Shared-Memory Clusters**. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 751–758, 2012. 129

[29] Daniel Cabrera, Xavier Martorell, Georgi Gaydadjiev, Eduard Ayguade, and Daniel Jiménez-González. **OpenMP extensions for FPGA accelerators**. In *Proceedings of the 9th international conference on Systems, architectures, modeling and simulation*, SAMOS'09, pages 17–24, Piscataway, NJ, USA, 2009. IEEE Press. [Online] http://dl.acm.org/citation.cfm?id=1812707.1812714. 129

[30] Calypto Design Systems Inc. **Catapult Family**. [Online] http://calypto.com/en/products/catapult/overview, 2013. 3

[31] Olivier Certner, Zheng Li, Pierre Palatin, Olivier Temam, Frederic Arzel, and Nathalie Drach. **A practical approach for reconciling high and predictable performance in non-regular parallel programs**. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 740–745, New York, NY, USA, 2008. ACM. [Online] http://doi.acm.org/10.1145/1403375.1403555. 60

[32] Rohit Chandra, Ding-Kai Chen, Robert Cox, Dror E. Maydan, Nenad Nedeljkovic, and Jennifer M. Anderson. **Data distribution support on distributed shared memory multiprocessors**. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, PLDI '97, pages 334–345, New York, NY, USA, 1997. ACM. [Online] http://doi.acm.org/10.1145/258915.258945. 45, 60, 62

[33] B. Chapman, Lei Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and Alan Gatherer. **Implementing OpenMP on a high performance embedded multicore MPSoC**. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8, 2009. 28, 31, 34

[34] Barbara M. Chapman, F. Bregier, Amit Patil, and Achal Prabhakar. **Achieving performance under OpenMP on cc-NUMA and software distributed shared memory systems.** *Concurrency and Computation: Practice and Experience*, **14**(8-9):713–739, 2002. [Online] http://dblp.uni-trier.de/db/journals/concurrency/concurrency14.html#ChapmanBPP02. 45, 60

[35] Jongsok Choi, K. Nam, A. Canis, J. Anderson, S. Brown, and T. Czajkowski. **Impact of Cache Architecture and Interface on Performance and Area of FPGA-Based Processor/Parallel-Accelerator Systems**. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 17–24, 2012. 2

[36] E.S. Chung, P.A. Milder, J.C. Hoe, and Ken Mai. **Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs?** In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 225–236, 2010. 3

[37] J. Cong, M.A. Ghodrat, M. Gill, B. Grigorian, and G. Reinman. **Architecture support for accelerator-rich CMPs**. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pages 843–849, 2012. 129, 146

[38] Marco Cornero and Anyuru Andreas. **Multiprocessing in Mobile Platforms: the Marketing and the Reality**. [Online]http://www.electronics-eetimes.com/en/multiprocessing-in-mobiles-platforms-the-marketing-and-the-reality.html?\cmp_id=34&news_id=222915456, 2013. 1

[39] M. Dehyadegari, A. Marongiu, M.R. Kakoee, L. Benini, S. Mohammadi, and N. Yazdani. **A tightly-coupled multi-core cluster with shared-memory HW accelerators**. In *Embedded Computer Systems (SAMOS), 2012 International Conference on*, pages 96–103, 2012. 148

[40] Vassilios V. Dimakopoulos, Panagiotis E. Hadjidoukas, and Giorgos Ch. Philos. **A microbenchmark study of OpenMP overheads under nested parallelism**. In *Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, IWOMP'08, pages 1–12, Berlin, Heidelberg, 2008. Springer-Verlag. [Online] http://dl.acm.org/citation.cfm?id=1789826.1789828. 85, 94

[41] A.J. Dorta, C. Rodriguez, and F. de Sande. **The OpenMP source code repository**. In *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*, pages 244–250, 2005. 48

[42] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. **Evaluation of OpenMP task scheduling strategies**. In *Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, IWOMP'08, pages 100–110, Berlin, Heidelberg, 2008. Springer-Verlag. [Online] http://dl.acm.org/citation.cfm?id=1789826.1789838. 5, 21, 86, 87, 100

[43] P. Dziurzanski, W. Bielecki, K. Trifunovic, and M. Kleszczonek. **A system for transforming an ANSI C code with OpenMP directives into a SystemC description**. In *Design and Diagnostics of Electronic Circuits and systems, 2006 IEEE*, pages 151–152, 2006. 130

[44] J. Eker and J. W. Janneck. **CAL Language Report Specification of the CAL Actor Language**. Technical Report UCB/ERL M03/48, EECS Department, University of California, Berkeley, 2003. [Online] http://www.eecs.berkeley.edu/Pubs/TechRpts/2003/4186.html. 2

[45] Magnus Ekman, Fredrik Warg, and Jim Nilsson. **An in-depth look at computer performance growth**. *SIGARCH Comput. Archit. News*, **33**(1):144–147, March 2005. [Online] http://doi.acm.org/10.1145/1055626.1055646. 1

[46] Element CXI. **ECA-64 elemental computing array**. [Online] http://www.elementcxi.com/downloads/ECA64ProductBrief.doc, 2008. 25

[47] P.G. Emma and E Kursun. **Is 3D chip technology the next growth engine for performance improvement?** *IBM Journal of Research and Development*, **52**(6):541–552, 2008. 58, 60

[48] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. **Neural Acceleration for General-Purpose Approximate Programs**. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pages 449–460, 2012. 3, 162

[49] Marco Facchini, Trevor Carlson, Anselme Vignon, Martin Palkovic, Francky Catthoor, Wim Dehaene, Luca Benini, and Paul Marchal. **System-level power/performance evaluation of 3D stacked DRAMs for mobile applications**. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 923–928, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association. [Online] http://dl.acm.org/citation.cfm?id=1874620.1874847. 58, 60

[50] C.F. Fajardo, Zhen Fang, R. Iyer, G.F. Garcia, Seung Eun Lee, and Li Zhao. **Buffer-Integrated-Cache: A cost-effective SRAM architecture for handheld and embedded platforms**. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 966–971, 2011. 128, 142

[51] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. **Sequoia: programming the memory hierarchy**. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. [Online] http://doi.acm.org/10.1145/1188455.1188543. 14

[52] FSF - The GNU Project. **GOMP - An OpenMP implementation for GCC**. [Online] http://gcc.gnu.org/projects/gomp/. 14, 16, 26, 31, 69, 74, 86, 87, 109, 113, 119, 122, 123, 124, 137, 161

[53] Luo-feng Geng, Duo-li Zhang, and Ming-Lun Gao. **Performance evaluation of cluster-based homogeneous multiprocessor system-on-chip using FPGA device**. In *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*, **4**, pages V4–144–V4–147, 2010. 27

[54] Luo-Feng Geng, Duo-li Zhang, Ming-Lun Gao, Ying-Chun Chen, and Gao-Ming Du. **Prototype design of cluster-based homogeneous Multiprocessor System-on-Chip**. In *Anti-counterfeiting, Security, and Identification in Communication, 2009. ASID 2009. 3rd International Conference on*, pages 311–315, 2009. 25, 27

[55] S. Gochman, A. Mendelson, A. Naveh, and E. Rotem. **Introduction to Intel Core Duo processor architecture**. *Intelligence/sigart Bulletin*, 2006. 1

[56] Marc González, José Oliver, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, and Nacho Navarro. **OpenMP Extensions for Thread Groups and Their Run-Time Support**. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, LCPC '00, pages 324–338, London, UK, 2001. Springer-Verlag. [Online] http://dl.acm.org/citation.cfm?id=645678.663945. 86

[57] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. **A high-performance, portable implementation of the MPI message passing interface standard**, September 1996. [Online] http://dx.doi.org/10.1016/0167-8191(96)00024-5. 27

[58] S.Q. Gu, P. Marchal, M. Facchini, F. Wang, M. Suh, D. Lisk, and M. Nowak. **Stackable memory of 3D chip integration for mobile applications**. In *Electron Devices Meeting, 2008. IEDM 2008. IEEE International*, pages 1–4, 2008. 60

[59] Panagiotis Hadjidoukas and Vassilios Dimakopoulos. **Nested Parallelism in the OMPi OpenMP/C Compiler**. In Anne-Marie Kermarrec, Luc Boug, and Thierry Priol, editors, *Euro-Par 2007 Parallel Processing*, **4641** of *Lecture Notes in Computer Science*, pages 662–671. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-74466-5_70. [Online] http://dx.doi.org/10.1007/978-3-540-74466-5_70. 85

[60] Thomas Hall and Ken Kent. **A Hardware/Software Co-specification Methodology for Multiple Processor Custom Hardware Devices Based On OpenMP**. In *Proceedings of the 2008 Research Exposition*, 2008. 129

[61] Debra Hensgen, Raphael Finkel, and Udi Manber. **Two algorithms for barrier synchronization**. *Int. J. Parallel Program.*, **17**(1):1–17, February 1988. [Online] http://dx.doi.org/10.1007/BF01379320. 39

[62] J. Holt, A. Agarwal, S. Brehmer, M. Domeika, P. Griffin, and F. Schirrmeister. **Software Standards for the Multicore Era**. *Micro, IEEE*, **29**(3):40–51, 2009. 2

[63] Intel Corporation. **Threading Building Blocks**. [Online] http://threadingbuildingblocks.org/, 2006. 2, 3, 14, 120, 121

[64] Intel Corporation. **Intel Xeon Phi**. [Online] http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html, 2012. 1

[65] Intel Inc. **Intel Xeon Phi**, 2013. [Online] http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html. 129, 146

[66] Dongsuk Jeon, Yejoong Kim, Inhee Lee, Zhengya Zhang, David Blaauw, and Dennis Sylvester. **A LOW-POWER VGA FULL-FRAME FEATURE EXTRACTION PROCESSOR**. 3

[67] Woo-Chul Jeun and Soonhoi Ha. **Effective OpenMP Implementation and Translation For Multiprocessor System-On-Chip without Using OS**. In *Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific*, pages 44–49, 2007. 28, 33, 34

[68] Xin Jin, Yukun Song, and Duoli Zhang. **FPGA prototype design of the computation nodes in a cluster based MPSoC**. In *Anti-Counterfeiting Security and Identification in Communication (ASID), 2010 International Conference on*, pages 71–74, 2010. 25

[69] J. Joven, A. Marongiu, F. Angiolini, L. Benini, and G. De Micheli. **Exploring programming model-driven QoS support for NoC-based platforms**. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 65–74, 2010. 28

[70] Kalray Corporation. **Many-core Kalray MPPA**. [Online] http://www.kalray.eu/, 2012. 1, 2, 7, 12, 25

[71] Uksong Kang, Hoe-Ju Chung, Seongmoo Heo, Soon-Hong Ahn, Hoon Lee, Soo-Ho Cha, Jaesung Ahn, DukMin Kwon, Jin-Ho Kim, Jae-Wook Lee, Han-Sung Joo, Woo-Seop Kim, Hyun-Kyung Kim, Eun-Mi Lee, So-Ra Kim, Keum-Hee Ma, Dong-Hyun Jang, Nam-Seog Kim, Man-Sik Choi, Sae-Jang Oh, Jung-Bae Lee, Tae-Kyung Jung, Jei-Hwan Yoo, and Changhyun Kim. **8Gb 3D DDR3 DRAM using through-silicon-via technology**. In *Solid-State Circuits Conference - Digest of Technical Papers, 2009. ISSCC 2009. IEEE International*, pages 130–131,131a, 2009. 59

[72] L.J. Karam, I. AlKamal, Alan Gatherer, G.A. Frantz, D.V. Anderson, and B.L. Evans. **Trends in multicore DSP platforms**. *Signal Processing Magazine, IEEE*, **26**(6):38–49, 2009. 26

[73] Sven Karlsson. **A portable and efficient thread library for OpenMP**. In *In Proc. 6th European Workshop on OpenMP, KTH Royal Institute of Technology*, pages 43–47. John Wiley, 2004. 85

[74] Karonis, Nicholas T. and Toonen, Brian and Foster, Ian. **MPICH-G2: a Grid-enabled implementation of the Message Passing Interface**. *J. Parallel Distrib. Comput.*, **63**(5):551–563, May 2003. [Online] http://dx.doi.org/10.1016/S0743-7315(03)00002-9. 27

[75] M. Kawano, N. Takahashi, Y. Kurita, K. Soejima, M. Komuro, and S. Matsui. **Three-Dimensional Packaging Technology for Stacked DRAM With 3-Gb/s Data Transfer**. *Electron Devices, IEEE Transactions on*, **55**(7):1614–1620, 2008. 59

[76] Taeho Kgil, Ali Saidi, Nathan Binkert, Steve Reinhardt, Krisztian Flautner, and Trevor Mudge. **PicoServer: Using 3D stacking technology to build energy efficient servers**. *J. Emerg. Technol. Comput. Syst.*, **4**(4):16:1–16:34, November 2008. [Online] http://doi.acm.org/10.1145/1412587.1412589. 58, 59

[77] Thilo Kielmann, Rutger F. H. Hofman, Henri E. Bal, Aske Plaat, and Raoul A. F. Bhoedjang. **MagPIe: MPIs Collective Communication Operations for Clustered Wide Area Systems**. In *Proc PPoPP'99*, pages 131–140, 1999. 27

[78] Theo Kluter, Philip Brisk, Paolo Ienne, and Edoardo Charbon. **Speculative DMA for architecturally visible storage in instruction set extensions**. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, CODES+ISSS '08, pages 243–248, New York, NY, USA, 2008. ACM. [Online] http://doi.acm.org/10.1145/1450135.1450191. 128

[79] Theo Kluter, Philip Brisk, Paolo Ienne, and Edoardo Charbon. **Way Stealing: cache-assisted automatic instruction set extensions**. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 31–36, New York, NY, USA, 2009. ACM. [Online] http://doi.acm.org/10.1145/1629911.1629923. 128

[80] Kronos Group. **The OpenCL 1.1 Specifications**. [Online] http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf, 2010. 4, 13, 126, 129

[81] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. **Carbon: architectural support for fine-grained parallelism on chip multiprocessors**. *SIGARCH Comput. Archit. News*, **35**:162–173, June 2007. [Online] http://doi.acm.org/10.1145/1273440.1250683. 86, 87

[82] G. Kyriazis. **Heterogeneous System Architecture: A Technical Review**, 2012. [Online] http://developer.amd.com/wordpress/media/2012/10/hsa10.pdf. 3, 129

[83] James Larus. **Spending Moore's dividend**. *Commun. ACM*, **52**(5):62–69, May 2009. [Online] http://doi.acm.org/10.1145/1506409.1506425. 1

[84] Thai Hoang Le and Len Tien Bui. **An approach to combine AdaBoost and Artificial Neural Network for detecting human faces**. In *Systems, Man and Cybernetics, 2008. SMC 2008. IEEE International Conference on*, pages 3411–3416, 2008. 162

[85] Y. Y. Leow, C. Y. Ng, and W.F. Wong. **Generating hardware from OpenMP programs**. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 73–80, 2006. 130

[86] Bin Li, Zhen Fang, and R. Iyer. **Template-based memory access engine for accelerators in SoCs**. In *Design Automation Conference (ASP-DAC), 2011 16th Asia and South Pacific*, pages 147–153, 2011. 128

[87] Feihul Li, C. Nicopoulos, T. Richardson, Yuan Xie, V. Narayanan, and M. Kandemir. **Design and Management of 3D Chip Multiprocessors Using Network-in-Memory**. In *Computer Architecture, 2006. ISCA '06. 33rd International Symposium on*, pages 130–141, 2006. 60

[88] Xueliang Li, Guihai Yan, Yinhe Han, and Xiaowei Li. **SmartCap: User experience-oriented power adaptation for smartphone's application processor**. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 57–60, 2013. 1, 3

[89] Linus Torvalds. **The Linux "Vanilla" Kernel**. [Online] https://www.kernel.org/, 2013. 5

[90] F. Liu and V. Chaudhary. **Extending OpenMP for heterogeneous chip multiprocessors**. In *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 161–168, 2003. 28, 33

[91] Feng Liu and Vipin Chaudhary. **A practical OpenMP compiler for system on chips**. In *Proceedings of the OpenMP applications and tools 2003 international conference on OpenMP shared memory parallel programming*, WOMPAT'03, pages 54–68, Berlin, Heidelberg, 2003. Springer-Verlag. [Online] http://dl.acm.org/citation.cfm?id=1761900.1761907. 28, 33

[92] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon. **Analyzing on-chip communication in a MPSoC environment**. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, **2**, pages 752–757 Vol.2, 2004. 48

[93] G.H. Loh. **3D-Stacked Memory Architectures for Multi-core Processors**. In *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pages 453–464, 2008. 58, 60, 61

[94] A. Marongiu and L. Benini. **An OpenMP Compiler for Efficient Use of Distributed Scratchpad Memory in MPSoCs**. *Computers, IEEE Transactions on*, **61**(2):222–236, 2012. 45, 46, 47, 48, 66, 74

[95] Xavier Martorell, Eduard Ayguad, Nacho Navarro, Julita Corbaln, Marc Gonzlez, and Jess Labarta. **Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors**. In *in NUMA Multiprocessors. In 13th Int. Conference on Supercomputing ICS'99, Rhodes*, pages 294–301, 1999. 85

[96] Massachusets Institute of Technology. **The Cilk Project**. [Online] http://supertech.csail.mit.edu/cilk/, 1998. 2, 3, 14, 86, 121

[97] Ramachandra Nanjegowda, Oscar Hernandez, Barbara Chapman, and Haoqiang H. Jin. **Scalability Evaluation of Barrier Algorithms for OpenMP**. In *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, IWOMP '09, pages 42–52, Berlin, Heidelberg, 2009. Springer-Verlag. [Online] http://dx.doi.org/10.1007/978-3-642-02303-3_4. 5, 38

[98] Girija J. Narlikar and Guy E. Blelloch. **Space-Efficient Scheduling of Nested Parallelism**. *ACM Transactions on Programming Languages and Systems*, **21**, 1999. 85

[99] S. Niar, S. Meftali, and J. Dekeyser. **Power consumption awareness in cache memory design with SystemC**. In *Microelectronics, 2004. ICM 2004 Proceedings. The 16th International Conference on*, pages 244–247, 2004. 2

[100] Dan Nicolaescu, Alex Veidenbaum, and Alex Nicolau. **Reducing data cache energy consumption via cached load/store queue**. In *Proceedings of the 2003 international symposium on Low power electronics and design*, ISLPED '03, pages 252–257, New York, NY, USA, 2003. ACM. [Online] http://doi.acm.org/10.1145/871506.871569. 2

[101] NVIDIA. **Next Generation CUDA Compute Architecture: Fermi - WhitePaper**. [Online] http://www.nvidia.fr/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2010. 1, 7, 25, 142

[102] OAR Corporation. **Real-Time Executive for Multiprocessor Systems**. [Online] http://www.rtems.com/, 2006. 12

[103] Y. Ojima, M. Sato, H. Harada, and Y. Ishikawa. **Performance of cluster-enabled OpenMP for the SCASH software distributed shared memory system**. In *Cluster Computing and the Grid, 2003. Proceedings. CCGrid 2003. 3rd IEEE/ACM International Symposium on*, pages 450–456, 2003. 28

[104] Open Source Computer Vision library. [Online] http://opencv.willowgarage.com/wiki/. 129

[105] OpenAcc Specifications 1.0, 2011. [Online] http://www.openacc.org/sites/default/files/OpenACC.1.0_0.pdf. 4, 157, 162

[106] OpenMP Architecture Review Board. **OpenMP Application Program Interface v3.1**. [Online] http://www.openmp.org/mp-documents/OpenMP3.1.pdf, 2011. 2, 3, 12, 14, 15, 23, 86, 113, 119, 121, 161

[107] OpenMP Architecture Review Board. **OpenMP Application Program Interface v4**. [Online] http://www.openmp.org/mp-documents/OpenMP3.1.pdf, 2011. 4, 127, 136, 162

[108] Ozcan Ozturk, Feng Wang, Mahmut T. Kandemir, and Yuan Xie. **Optimal topology exploration for application-specific 3D architectures**. In *ASP-DAC*, pages 390–395, 2006. 60

[109] Plurality Ltd. **The HyperCore Processor**. [Online] http://www.plurality.com/hypercore.html. xvi, 1, 2, 7, 12, 14, 25, 121, 146, 147

[110] Plurality Ltd. **The HyperCore Processor**. [Online] www.plurality.com/hypercore.html. 87

[111] Abbas Rahimi, Igor Loi, Mohammad Reza Kakoee, and Luca Benini. **A fully-synthesizable single-cycle interconnection network for Shared-L1 processor clusters**. In *Proceedings of the Design, Automation Test in Europe Conference Exhibition (DATE) 2011*, pages 1 – 6, 2011. xvi, 87, 147

[112] Edward Rosten, R. Porter, and Tom Drummond. **Faster and Better: A Machine Learning Approach to Corner Detection**. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, **32**(1):105–119, 2010. 111

[113] Mitsuhisa Sato, Hiroshi Harada, Atsushi Hasegawa, and Yutaka Ishikawa. **Cluster-enabled OpenMP: An OpenMP compiler for the SCASH software distributed shared memory system**. *Sci. Program.*, **9**(2,3):123–130, August 2001. [Online] http://dl.acm.org/citation.cfm?id=1239928.1239934. 60

[114] M. Shafiq, M. Pericas, N. Navarro, and E. Ayguade. **TARCAD: A template architecture for reconfigurable accelerator designs**. In *Application Specific Processors (SASP), 2011 IEEE 9th Symposium on*, pages 8–15, 2011. 3

[115] Amar Shan. **Heterogeneous processing: a strategy for augmenting moore's law**. *Linux J.*, **2006**(142):7, 2006. [Online] http://portal.acm.org/citation.cfm?id=1119128.1119135. 3

[116] Silicon Graphics International Corporation. **SGI Origin 3000**. http://www.sgi.com/products/remarketed/servers/origin3000.html, 2009. 62

[117] Herb Sutter. **The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software**. *Dr. Dobb's Journal*, **30**(3):202–210, 2005. [Online] http://www.gotw.ca/publications/concurrency-ddj.htm. 1

[118] S. Swanson and M.B. Taylor. **Greendroid: Exploring the next evolution in smartphone application processors**. *Communications Magazine, IEEE*, **49**(4):112–119, 2011. 1, 3, 129

[119] Synopsys Inc. **Design Compiler Graphical**. [Online] http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DCGraphical/Pages/default.aspx. 154

[120] Synopsys Inc. **Processor Designer**. [Online] http://www.synopsys.com/Systems/BlockDesign/ProcessorDev/Pages/default.aspx. 129

[121] Yoshizumi Tanaka, Kenjiro Taura, Mitsuhisa Sato, and Akinori Yonezawa. **Performance Evaluation of OpenMP Applications with Nested Parallelism**. In *Selected Papers from the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, LCR '00, pages 100–112, London, UK, 2000. Springer-Verlag. [Online] http://dl.acm.org/citation.cfm?id=648049.761156. 85, 86

[122] Tensilica. **Xtensa Customizable Processors**. [Online] http://www.tensilica.com/products/xtensa-customizable.htm. 129

[123] Xavier Teruel, Xavier Martorell, Alejandro Duran, Roger Ferrer, and Eduard Ayguadé. **Support for OpenMP tasks in Nanos v4**. In *Proceedings of the 2007 conference of the center for advanced studies on Collaborative research*, CASCON '07, pages 256–259, Riverton, NJ, USA, 2007. IBM Corp. [Online] http://dx.doi.org/10.1145/1321211.1321241. 14

[124] Texas Instruments. **TMS320TCI6488 DSP Platform**. [Online] http://focus.ti.com/lit/ml/sprt415/sprt415.pdf. 30

[125] Texas Instruments. **TNETV3020 carrier infrastructure platform**. [Online] http://focus.ti.com/lit/ml/spat174a/spat174a.pdf. 30

[126] The Eclipse Foundation. **Eclipse IDE**. [Online] http://www.eclipse.org/, 2013. 12

[127] The Khronos Group. [Online] http://www.khronos.org/. 4, 129

[128] The Khronos Group. **OpenVX**, 2013. [Online] http://www.khronos.org/openvx. 4, 129, 146, 162

[129] The MPI Forum. **MPI Standard 3.0**. [Online] http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf. 27

[130] The Portland Group. **PGI Acceleration Programming Model for Fortran & C**. [Online] http://www.pgroup.com/lit/whitepapers/pgi_accel_prog_model_1.3.pdf, 2010. 129

[131] Daouda Traoré, Jean-Louis Roch, Nicolas Maillard, Thierry Gautier, and Julien Bernard. **Deque-Free Work-Optimal Parallel STL Algorithms**. In *Proceedings of the 14th international Euro-Par conference on Parallel Processing*, Euro-Par '08, pages 887–897, Berlin, Heidelberg, 2008. Springer-Verlag. [Online] http://dx.doi.org/10.1007/978-3-540-85451-7_95. 60

[132] A. Tumeo, M. Monchiero, G. Palermo, F. Ferrandi, and D. Sciuto. **Lightweight DMA management mechanisms for multiprocessors on FPGA**. In *Application-Specific Systems, Architectures and Processors, 2008. ASAP 2008. International Conference on*, pages 275–280, 2008. 155

[133] Université de Bretagne-Sud. **GAUT HLS Tool**. [Online] http://www-labsticc.univ-ubs.fr/www-gaut/publications/gaut_flyer.pdf, 1993. 3, 138, 156, 159

[134] University of Edinburgh. **OpenMP Microbenchmarks V2.0**. [Online] http://www2.epcc.ed.ac.uk/computing/research_activities/openmpbench/openmp_index.html. 94

[135] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. **Conservation cores: reducing the energy of mature computations**. *SIGARCH Comput. Archit. News*, **38**(1):205–218, March 2010. [Online] http://doi.acm.org/10.1145/1735970.1736044. 129

[136] Oreste Villa, Gianluca Palermo, and Cristina Silvano. **Efficiency and scalability of barrier synchronization on NoC based many-core architectures**. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '08, pages 81–90, New York, NY, USA, 2008. ACM. [Online] http://doi.acm.org/10.1145/1450095.1450110. 38, 41

[137] P. Viola and M. Jones. **Rapid object detection using a boosted cascade of simple features**. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, **1**, pages I–511–I–518 vol.1, 2001. 154, 157

[138] C. Weis, I. Loi, L. Benini, and N. Wehn. **An energy efficient DRAM subsystem for 3D integrated SoCs**. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 1138–1141, 2012. 3, 58

[139] Barry Wilkinson and Michael Allen. *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999. 38

[140] Hae woo Park, Hyunok Oh, and Soonhoi Ha. **Multiprocessor SoC design methods and tools**. *Signal Processing Magazine, IEEE*, **26**(6):72–79, 2009. 26

[141] Wm. A. Wulf and Sally A. McKee. **Hitting the memory wall: implications of the obvious**. *SIGARCH Comput. Archit. News*, **23**(1):20–24, March 1995. 1

[142] Xilinx Inc. **Zynq Series**. [Online] http://www.xilinx.com/content/xilinx/en/products/silicon-devices/soc/zynq-7000.html. 125

[143] L. Xue, M. Kandemir, G. Chen, F. Li, O. Ozturk, R. Ramanarayanan, and B. Vaidyanathan. **Locality-Aware Distributed Loop Scheduling for Chip Multiprocessors**. In *Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference: Embedded Systems*, VLSID '07, pages 251–258, Washington, DC, USA, 2007. IEEE Computer Society. [Online] http://dx.doi.org/10.1109/VLSID.2007.97. 60

[144] Liu Zhuo, Gaoming Du, Duoli Zhang, Yukun Song, Li Li, and Hongbin Pan. **Design and implemetion of DDR2 wrapper for cluster based MPSoC**. In *Anti-Counterfeiting Security and Identification in Communication (ASID), 2010 International Conference on*, pages 60–62, 2010. 25

[145] C. Zinner and W. Kubinger. **ROS-DMA: A DMA Double Buffering Method for Embedded Image Processing with Resource Optimized Slicing**. In *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, pages 361–372, 2006. 155