Università degli Studi di Bologna

FACOLTA' DI INGEGNERIA

DOTTORATO DI RICERCA IN INGEGNERIA ELETTRONICA,
INFORMATICA E DELLE TELECOMUNICAZIONI
Ciclo XXII

TECNICHE DI OTTIMIZZAZIONE DEL SOFTWARE PER SISTEMI SU SINGOLO CHIP PER APPLICAZIONI DI NOMADIC COMPUTING

Tesi di Dottorato di:
ANDREA MARONGIU

Relatori:
Chiar.mo Prof. **LUCA BENINI**

Coordinatore:
Chiar.mo Prof. **PAOLA MELLO**

Settore Scientifico Disciplinare: ING/INF01 Elettronica
Anno Accademico 2008/09

Tecniche di Ottimizzazione del Software per Sistemi su Singolo Chip per Applicazioni di Nomadic Computing

Andrea Marongiu
Department of Electronic, Computer Science and Systems
University of Bologna

A thesis submitted for the degree of
PhilosophiæDoctor (PhD)

March 2010

Abstract

I moderni sistemi embedded sono equipaggiati con risorse hardware che consentono l'esecuzione di applicazioni molto complesse come il decoding audio e video. La progettazione di simili sistemi deve soddisfare due esigenze opposte. Da un lato è necessario fornire un elevato potenziale computazionale, dall'altro bisogna rispettare dei vincoli stringenti riguardo il consumo di energia. Uno dei trend più diffusi per rispondere a queste esigenze opposte è quello di integrare su uno stesso chip un numero elevato di processori caratterizzati da un design semplificato e da bassi consumi. Tuttavia, per sfruttare effettivamente il potenziale computazionale offerto da una batteria di processori è necessario rivisitare pesantemente le metodologie di sviluppo delle applicazioni. Con l'avvento dei sistemi multi-processore su singolo chip (MPSoC) il *parallel programming* si è diffuso largamente anche in ambito embedded. Tuttavia, i progressi nel campo della programmazione parallela non hanno mantenuto il passo con la capacità di integrare hardware parallelo su un singolo chip.

Oltre all'introduzione di multipli processori, la necessità di ridurre i consumi degli MPSoC comporta altre soluzioni architetturali che hanno l'effetto diretto di complicare lo sviluppo delle applicazioni. Il design del sottosistema di memoria, in particolare, è un problema critico. Integrare sul chip dei banchi di memoria consente dei tempi d'accesso molto brevi e dei consumi molto contenuti. Sfortunatamente, la quantità di memoria on-chip che può essere integrata in un MPSoC è molto limitata. Per questo motivo è necessario aggiungere dei banchi di memoria off-chip, che hanno una capacità molto maggiore, come maggiori sono i consumi e i tempi d'accesso. La maggior parte degli MPSoC attualmente in commercio destina una parte del budget di area all'implementazione di memorie cache e/o scratchpad.

Le scratchpad (SPM) sono spesso preferite alle cache nei sistemi MPSoC embedded, per motivi di maggiore predicibilità, minore occupazione d'area e – soprattutto – minori consumi. Per contro, mentre l'uso delle cache è completamente trasparente al programmatore, le SPM devono essere esplicitamente gestite dall'applicazione.

Esporre l'organizzazione della gerarchia di memoria all'applicazione consente di sfruttarne in maniera efficiente i vantaggi (ridotti tempi d'accesso e consumi). Per contro, per ottenere questi benefici è necessario scrivere le applicazioni in maniera tale che i dati vengano partizionati e allocati sulle varie memorie in maniera opportuna. L'onere di questo compito complesso ricade ovviamente sul programmatore. Questo scenario descrive bene l'esigenza di modelli di programmazione e strumenti di supporto che semplifichino lo sviluppo di applicazioni parallele.

In questa tesi viene presentato un framework per lo sviluppo di software per MPSoC embedded basato su OpenMP. OpenMP è uno standard di fatto per la programmazione di multiprocessori con memoria shared, caratterizzato da un semplice approccio alla parallelizzazione tramite annotazioni (direttive per il compilatore). La sua interfaccia di programmazione consente di esprimere in maniera naturale e molto efficiente il parallelismo a livello di loop, molto diffuso tra le applicazioni embedded di tipo *signal processing* e *multimedia*.

OpenMP costituisce un ottimo punto di partenza per la definizione di un modello di programmazione per MPSoC, soprattutto per la sua semplicità d'uso. D'altra parte, per sfruttare in maniera efficiente il potenziale computazionale di un MPSoC è necessario rivisitare profondamente l'implementazione del supporto OpenMP sia nel compilatore che nell'ambiente di supporto a runtime. Tutti i costrutti per gestire il parallelismo, la suddivisione del lavoro e la sincronizzazione inter-processore comportano un costo in termini di overhead che deve essere minimizzato per non comprometterne i vantaggi della parallelizzazione. Questo può essere ottenuto soltanto tramite una accurata analisi delle caratteristiche hardware e l'individuazione dei potenziali colli di bottiglia nell'architettura. Una implementazione del

task management, della sincronizzazione a *barriera* e della condivisione dei dati che sfrutti efficientemente le risorse hardware consente di ottenere elevate performance e scalabilità.

La condivisione dei dati, nel modello OpenMP, merita particolare attenzione. In un modello a memoria condivisa le strutture dati (array, matrici) accedute dal programma sono fisicamente allocate su una unica risorsa di memoria raggiungibile da tutti i processori. Al crescere del numero di processori in un sistema, l'accesso concorrente ad una singola risorsa di memoria costituisce un evidente collo di bottiglia. Per alleviare la pressione sulle memorie e sul sistema di connessione vengono da noi studiate e proposte delle tecniche di *partizionamento* delle strutture dati. Queste tecniche richiedono che una singola entità di tipo array venga trattata nel programma come l'insieme di tanti sotto-array, ciascuno dei quali può essere fisicamente allocato su una risorsa di memoria differente. Dal punto di vista del programma, indirizzare un array partizionato richiede che ad ogni accesso vengano eseguite delle istruzioni per ri-calcolare l'indirizzo fisico di destinazione. Questo è chiaramente un compito lungo, complesso e soggetto ad errori. Per questo motivo, le nostre tecniche di partizionamento sono state integrate nella l'interfaccia di programmazione di OpenMP, che è stata significativamente estesa. Specificamente, delle nuove *direttive* e *clausole* consentono al programmatore di annotare i dati di tipo array che si vuole partizionare e allocare in maniera distribuita sulla gerarchia di memoria. Sono stati inoltre sviluppati degli strumenti di supporto che consentono di raccogliere informazioni di *profiling* sul pattern di accesso agli array. Queste informazioni vengono sfruttate dal nostro compilatore per allocare le partizioni sulle varie risorse di memoria rispettando una relazione di affinità tra il task e i dati. Più precisamente, i passi di allocazione nel nostro compilatore assegnano una determinata partizione alla memoria scratchpad locale al processore che ospita il task che effettua il numero maggiore di accessi alla stessa.

Contents

List of Figures

LIST OF FIGURES

Chapter 1

Introduction

1.1 Background

The scaling limitations of uniprocessors have led to an industry-wide turn towards chip multiprocessor (CMP) systems. Today, with the rise of multicore processors, parallel computing is everywhere. Multicore architectures have quickly spread to all computing domains, from personal computers to high performance supercomputers to embedded systems (1). Focusing on the latter, advances in multicore technology have significantly increased the performance of embedded Multiprocessor Systems-on-Chip (MPSoCs). As more and more hardware functions are integrated on the same device, embedded applications are becoming extremely sophisticated (2). Unfortunately, knowledge and experience in parallel programming have not kept pace with the trend towards parallel hardware. Multicore architectures require parallel computation and explicit management of the memory hierarchy, both of which add programming complexity and are unfamiliar to most programmers. This increased complexity is making the production of software the critical path in embedded system development (3).

This scenario calls for programming models and tools that aim at facilitating software development for embedded MPSoCs. One dominant form of parallelism in the embedded systems domain (e.g. signal processing applications) is data-level parallelism, where the same instruction is performed on different pieces of data in parallel following the *Single Instruction Multiple Data* model of execution.

This kind of parallelism is typically found within loop nests in an application, and is amenable – to some extent – to automatic compiler parallelization. Compiler-based

approaches to parallel application development have the key benefit that no burden of parallelization is imposed on the programmer. On the other hand, the applicability of this approach is usually limited to a set of applications with evident data-parallel regions, or to signal processing applications whose behaviors are relatively static and easily analyzable with the data-flow analysis methods.

A widely adopted alternative approach is that of extending standard languages from the uniprocessor domain, such as C, with specific constructs to express parallelism. Language-extension approaches require that the programmer provides information on where and how to parallelize a program by means of annotations.

OpenMP (11) is a well-known example of language extension with annotations, which has recently gained much attention in the embedded MPSoC domain. OpenMP is a de-facto standard for shared memory parallel programming. It consists of a set of compiler directives, library routines and environment variables that provide a simple means to specify parallel execution within a sequential code. The adoption of OpenMP as a programming model for embedded application development has three main benefits:

1. It allows programmers to continue using their familiar programming model, to which it adds only a little overhead for the annotations.

2. OpenMP is very efficient at expressing loop-level parallelism, which is an ideal target for the considered class of embedded applications.

3. The OpenMP compiler is relieved from the burden of parallelism extraction and can focus on exploiting the specified parallelism according to the target platform.

The OpenMP standard is very mature, but since it was originally designed for Symmetric Multi Processors (SMP) it assumes a uniform shared memory. On the contrary, one of the distinctive features of embedded MPSoCs is their complex memory subsystem. The chip has a limited area budget for on-chip memory that must be augmented by bulk commodity off-chip memory (3) (4). On-chip memory is often implemented as a set of scratchpad memories (SPM), each of which is tightly coupled to a processing element. Most of today's state-of-the-art processors for mobile and embedded systems feature similar on-chip memory organization (5) (6) (7) (8) (9) (10). A widespread memory model abstraction, which matches the described physical memory design, is

the Partitioned Global Address Space (PGAS). The PGAS model assumes a set of processors (nodes), each of which has its own local memory. Additionally, a single, globally addressable memory is provided from a portion of the local memory on each node. While accessing the global memory is more expensive than local memory access, it can be done without interacting with other nodes. Indeed – while they may be provided – explicit communication primitives (traditional *send* and *receive* operations) are unnecessary under the PGAS model since nodes may read and write shared data independently. Contention for access to global memory creates synchronization issues. A PGAS system must provide shared, atomic locking primitives to provide synchronization features (e.g. *test-and-set* semaphores).

Implementing the OpenMP memory model on top of a PGAS MPSoC is non trivial, and requires a careful design to take into account the NUMA organization of the memory hierarchy. Furthermore, the OpenMP execution model assumes homogeneous resources (processors and memories) when partitioning the workload among available threads. NUMA memory breaks this assumption, as accessing shared data may result in different access latencies from different threads. Array partitioning techniques are required to distribute shared data among appropriate memory segments in the PGAS. However, current programming languages and runtime systems do not provide the mechanisms necessary to efficiently exploit local memories to each core.

1.2 Thesis Contributions

In this thesis we first carry out a thorough study of the implementative challenges to support the SIMD/OpenMP parallel execution model on an embedded PGAS MPSoC. We then present an extended OpenMP API that augments the standard interface with features to expose the memory system at the application level. More specifically we make the following contributions.

- The design of API features to trigger data distribution and array partitioning, and their integration in the standard OpenMP programming interface
- The implementation of compiler support to instrument accesses to distributed arrays in the program with address translation routines for array partition localization in memory

- The implementation of a lightweight lookup mechanism based on compiler-generated metadata for low-cost array references
- The definition and implementation of allocation passes that exploit profile information on array access count to determine efficient placement of array partitions in memory

Moreover, the described techniques have been extensively evaluated on generic and representative PGAS MPSoC architectural templates. An initial evaluation of the applicability of array partitioning techniques to MPSoCs with vertically stacked (3D) memory is also provided.

1.3 Thesis Overview

We describe in this section the organization of the remainder of this thesis.

Automatic compiler parallelization identifies time-consuming loops and examines their dependencies to find out data-parallel regions. Parallel loop execution then leverages the SIMD model of computation, where the same loop code is assigned by the compiler to parallel threads which operate on separate portions of the data space. The SIMD execution model requires that threads synchronize on a global barrier after parallel region completion and prior to going ahead in the program. To achieve this goal compilers generate code that invokes the services of a runtime library where all of the support for thread creation, management and synchronization is implemented. The design of such a support library for an embedded shared memory MPSoC is presented in **Chapter 2**. Lightweight barrier implementation is also discussed and extensively evaluated, investigating the impact of library overhead on the parallelization granularity that can be exploited by the compiler.

OpenMP leverages similar kind of parallelism, but the approach is different. The programmer provides hints on where and how to parallelize the code, and the compiler focuses on generating optimized code for the specific platform. However, this goal cannot be achieved by a translator only, since OpenMP directives only allow to convey to the compiler high-level information about parallel program semantics. Most of the

target hardware specificities are enclosed within the OpenMP runtime environment, which is implemented as a library into which the compiler inserts explicit calls. The many peculiarities of MPSoC hardware call for a custom design of the runtime library. We present in **Chapter 3** the implementation of an OpenMP compiler and runtime environment for a generic embedded MPSoC template with explicitly managed memory hierarchy with NUMA organization. We found that an efficient exploitation of the memory hierarchy is key to achieving a scalable implementation of the OpenMP constructs.

Efficiently mapping loop-level parallelism in the OpenMP model requires parallel threads to execute on top of uniform resources. This assumption is not immediately applicable to PGAS MPSoCs, where accessing non-local memories is subject to NUMA latencies. To effectively address this issue it is necessary to partition shared array data. Each array slice must then be placed on the memory closest to the thread which has highest affinity with that slice. We describe in **Chapter 4** the design of suitable OpenMP extensions to trigger array partitioning, as well as the necessary compiler and runtime support. The techniques have been extensively evaluated on an embedded PGAS MPSoC with explicitly managed local SPMs.

In **Chapter 5** we discuss the applicability of array partitioning techniques to MPSoCs with vertically stacked DRAM memory. In the considered architectural template the effect of NUMA latencies is even more pronounced, which calls for revisited compiler support.

Finally, in **Chapter 6** we describe how our enhanced OpenMP programming framework can further be extended to deal with different sources of heterogeneity in MPSoCs, namely non-uniform processing resources due to core aging effects. We present techniques which leverage a partial recovery effect inherent in the considered aging phenomenon, Negative Bias Temperature Instability (NBTI), to schedule work to processor so as to maximize system lifetime.

Chapter 7 summarizes our achievements.

Bibliography

[1] G. Blake, R. Dreslinski, and T. Mudge, "A survey of multicore processors," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 26–37, November 2009. 1

[2] L. Karam, I. AlKamal, A. Gatherer, G. Frantz, D. Anderson, and B. Evans, "Trends in multicore dsp platforms," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 38–49, November 2009. 1

[3] H. woo Park, H. Oh, and S. Ha, "Multiprocessor soc design methods and tools," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 72–79, November 2009. 1, 2

[4] W. Wolf, "Multiprocessor system-on-chip technology," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 50–54, November 2009. 2

[5] Tilera Corp. Product Brief, "Tilepro64 processor," Available: http://www.tilera.com/pdf/ProductBrief_TILEPro64_Web_v2.pdf, 2008. 2

[6] Element CXI Inc. Product Brief, "Eca-64 elemental computing array," Available: http://www.elementcxi.com/downloads/ECA64ProductBrief.doc, 2008. 2

[7] Silicon Hive Databrief, "Hiveflex csp2000 series: Programmable ofdm communication signal processor," Available: http://www.siliconhive.com/Flex/Site/Page.aspx?PageID=8881, 2007. 2

[8] Arm Ltd. White Paper, "The arm cortex-a9 processors," Available: http://www.arm.com/pdfs/ARMCortexA-9Processors.pdf, 2007. 2

[9] NVIDIA Corp. CUDA Documentation, "Nvidia cuda: Compute unified device architecture," Available: http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf, 2008. 2

BIBLIOGRAPHY

[10] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementation: a performance view," *IBM J. Res. Dev.*, vol. 51, no. 5, pp. 559–572, 2007. 2

[11] www.openmp.org, "Openmp c and c++ application program interface v.3.0." 2

Chapter 2

Loop Parallelism and Barrier Synchronization

Many MPSoC applications are loop-intensive and amenable to automatic parallelization with suitable compiler support.

In this chapter we describe the role of the runtime support library to parallelizing compilers. Such a compiler is capable of extracting loop-level parallelism by assigning independent iterations to available threads. After proper loop analysis and transformation has been applied, the compiler generates code that synergistically interacts with the runtime library to orchestrate parallel execution.

One of the key components of any compiler-parallelized code is barrier instructions which are used to perform global synchronization across parallel threads. The runtime library also exposes synchronization facilities to the compiler. From the point of view of performance it is important that the cost for inter-processor synchronization is minimized to take advantage of the fine-grained parallelism enabled by compiler optimization.

2.1 Introduction

One of the key problems to be addressed in order to harness the potential computational power of embedded MPSoCs is *code parallelization*, which can be described as decomposing the application code into parallel threads which are then assigned to parallel cores for execution. Parallelizing an application requires skilled and knowledgeable

programmers, and involves several complex tasks such as concurrency extraction, data partitioning and code generation for thread management, synchronization and communication. Compiler support for automated code parallelization can be therefore very useful in practice.

Many array-intensive embedded applications are amenable to automatic parallelization with suitable compiler support, since most of the array processing kernels are typically implemented as a series of loop nests which the compiler can analyze.

One of the key components of any compiler-parallelized code is *barrier instructions*, which are used to perform global synchronization across parallel processors. As compared to programmer-parallelized codes, compiler-parallelized codes can contain larger number of barriers, mainly because a compiler has to be conservative in parallelizing an application (to preserve the original sequential semantics of the program), and this means, in most cases, inserting extra barrier instructions in the code.

Apart from the performance overheads they bring, barriers cause significant power consumption as well (1), and this power cost increases with the number of cores.

In the remainder of the chapter we describe the implementation of an MPSoC-suitable runtime support library targeted by a parallelizing compiler frontend. Particular emphasis is given to barrier implementation, and to the evaluation of synchronization cost induced by compiler-parallelized codes.

2.2 Background and Related work

Different schemes have been proposed for loop parallelization within different domains. In the context of high-end computing, fundamental relevant studies include (5) (6) (7) (8). Xue et al (9) explore a resource partitioning scheme for parallel applications in an MPSoC. Ozturk et al (10) propose a constraint network based approach to code parallelization for embedded MPSoCs, and Lee et al. (11) present a core mapping algorithm that addresses the problem of placing and routing the operations of a loop body. On the side of barrier synchronization many related work propose hybrid hardware-software approaches to achieve both fast synchronization and power savings. Liu et al (1) discuss an integrated hw/sw barrier mechanism that tracks the idle times spent by a processor waiting for other processors to get to the same point in the program. Using this knowledge they scale the frequency of the cores thus achieving power savings

without compromising the performances. Sampson et al (12) (13) present a mechanism for barrier synchronization on CMPs based on cache-lines invalidation. They ensure that all threads arriving at a barrier require an unavailable cache line to proceed, and, by placing additional hardware in the shared portions of the memory subsystem, they starve their requests until they all have arrived. Li and al (14) present a mixed hw/sw barrier mechanism to saving energy in parallel applications that exhibit barrier synchronization imbalance. Their approach transitions the processors arriving early at the barrier to a low power state, and wake them back up when the last processor gets there. Complete surveys on synchronization algorithms for Shared Memory Multiprocessors can be found in Kumar et al (15) and Mellor-Crummey (16).

2.3 Target Architecture

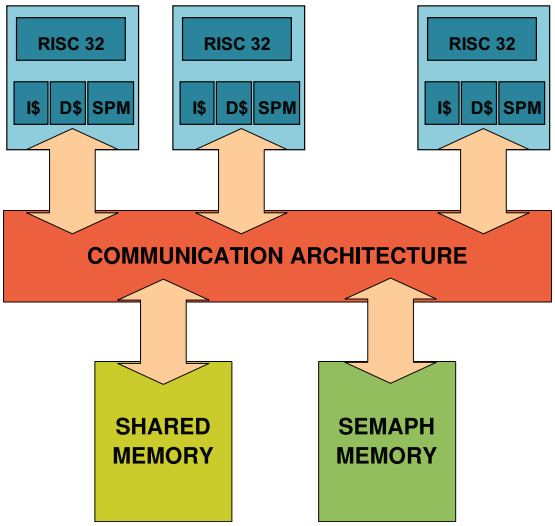


Figure 2.1: Shared memory architecture.

Our target MPSoC architectural template is depicted in Fig. 2.1. It consists of a configurable number of processing elements (PEs), based on a RISC-32 core and featuring on-tile instruction and data cache, plus scratchpad memory (SPM). Caches are globally non-coherent (see below). All cores can communicate through a shared memory device which is mapped in the global address space. Software synchronization primitives are built on top of a dedicated hardware semaphore device. This synchronization hardware can be viewed as a bank of memory mapped registers which are

accesses with *test-and-set* semantics. More specifically, reading from one location in this memory has the following semantic: reading a 0 from a register returns the value (*lock free*) and atomically updates the value to 1 (*lock acquired*).

In this architectural template, cache coherency is not supported in hardware. To guarantee data coherence from concurrent multiprocessor accesses shared memory can be configured to be non-cacheable but in this case it can only be inefficiently accessed by means of single transfers. Cacheability of the shared memory can be toggled, but in this case explicit software-controlled cache flush operations are needed.

2.4 Software Infrastructure

Figure 2.2 depicts our toolchain. The application input code is a sequential C program. As a first step, the program is processed by a parallelizing compiler frontend based on the SUIF technology. Here data dependency analysis is applied to determine which loops can be executed in parallel. After a parallel loop nest the compiler inserts a barrier instruction. This compiler-generated code relies on the synchronization features provided by our runtime library. The library is cross-compiled for the target architecture with a standard GCC compiler along with the transformed application code. The executable image is then loaded into our simulator.

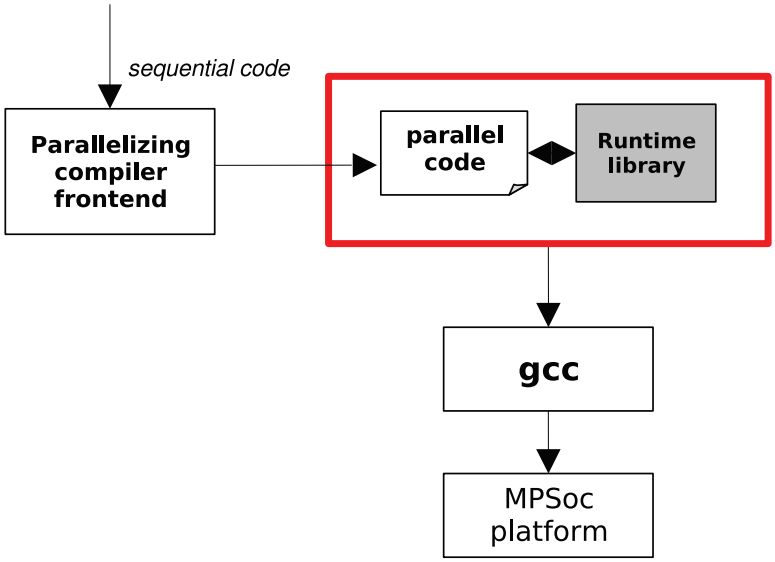


Figure 2.2: Compilation flow.

2.4.1 Parallelizing Compiler Front-End

A vast majority of data-intensive embedded applications are structured as a series of loop nests, each operating on large datasets. The most natural way of parallelizing such an application is to distribute loop iterations across parallel processors. The compiler front-end first extracts data dependencies; each data dependence can be represented using a vector, called the data dependence vector. When all data dependence vectors are considered together, the compiler figures out (conservatively) which loops in the nest can be executed in parallel. Specifically, a loop can be run parallel if it does not carry any data dependency.

Once candidate loop nests for parallel execution have been identified, the compiler generates code that partitions the workload assigned to each processor based on its physical ID, as shown in the green boxes in Fig. 2.3 (sequential loop → parallel loop). Beyond the end of the loop code the compiler inserts a global synchronization call. The purpose of this instruction is to prevent any processor to get ahead of the other processors and start executing the next piece of code in execution. Indeed, this may violate data dependences and ultimately change the original semantics of the application.

Figure 2.3 also describes how the parallelizing compiler alters the execution flow of the original sequential program to enable parallel execution. Loop code is extracted from its original location in the program and is moved into a compiler-generated function *(function outlining)*. Parallelized loop nests are replaced with calls to the `doall` function in the runtime environment.

2.4.2 Runtime Library

The support library orchestrates parallel execution by synchronizing code execution on the different cores. To keep the overhead for the execution of runtime services as small as possible the library is designed as a standalone (OS-less) middleware layer. A static task-to-processor mapping approach is adopted, where a single thread executes on each core.

The library implements the `main` function, which is executed by every processor. After a common initialization step, the processor with the highest ID is designated as the *controller*, while the other processors are promoted as *workers*, available for parallel computation.

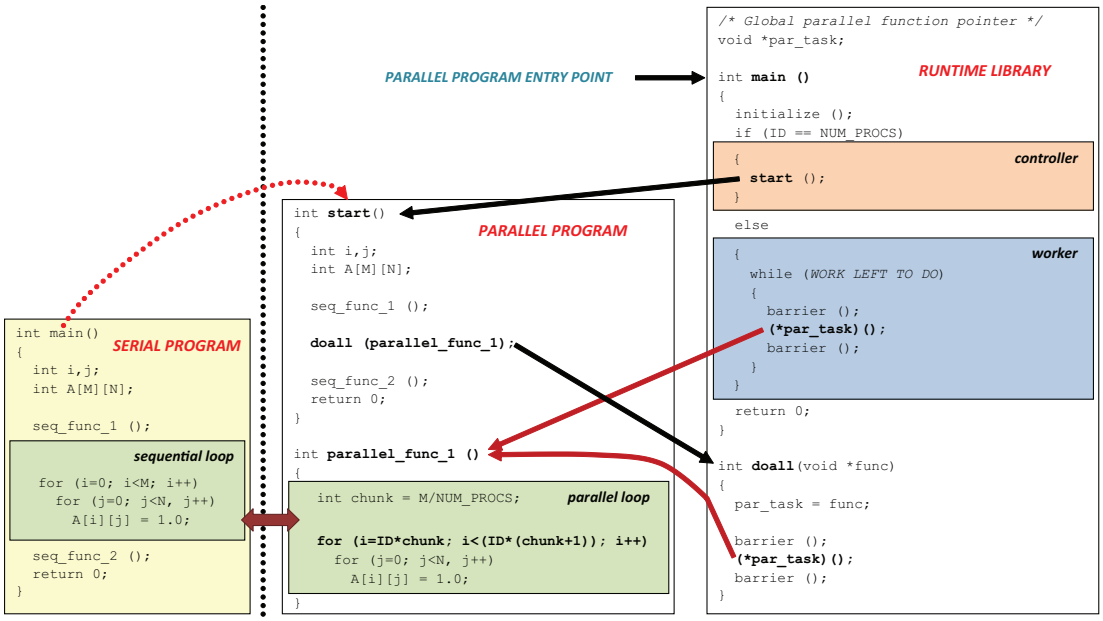


Figure 2.3: Original serial code and transformed parallel code. Interaction between parallel program and runtime library.

The `main` function in the original program is renamed by the compiler as `start`. After library initialization the execution flow on the *controller* jumps to `start` (red box), thus initiating target program execution, while *workers* enter a loop where they wait on a barrier for the *controller* to provide parallel work to do (blue box).

When a parallel loop is encountered by the *controller*, a call to `doall` is issued. Here the *controller* joins the barrier where *workers* are waiting, thus initiating parallel execution on every processor. The outlined functions containing the code of parallel loops are referenced by the library by means of a global function pointer.

2.4.3 Barrier Implementation

The library exports a set of synchronization primitives, namely locks (mutexes) and barriers, which are implemented on top of the *test-and-set* hardware semaphores described in Sec. 2.3. Mutex variables are instantiated as pointers to the semaphore memory. This allows to straightforwardly implement mutex *lock* and *unlock* as simple read/write operations.

```
/* Point to the base address of semaphore memory */
volatile char *mutex = SEMAPHORE_BASE;

/* Acquire lock */
void mutex_lock (int lock_ID)
{
  while (mutex[lock_ID]);
}

/* Release lock */
void mutex_unlock (int lock_ID)
{
  mutex[lock_ID] = 0;
}
```

A very simple and widely adopted barrier algorithm is the *centralized shared barrier*. This kind of barrier relies on shared entry and exit counters, which are atomically updated through lock-protected write operations.

```
  /* Increase ENTRY count */
  mutex_lock (ENTRY_LOCK_ID);
  entry_count++;
  mutex_unlock (ENTRY_LOCK_ID);

  /* Wait for all processors to enter */
  while (entry_count < NUM_PROCS) {}

  /* Increase EXIT count */
  mutex_lock (EXIT_LOCK_ID);
  exit_count++;
  /* Last arriving processor resets flags */
  if (exit_count == NUM_PROCS)
  {
    entry_count = 0;
    exit_count = 0;
  }
  mutex_unlock (EXIT_LOCK_ID);

  /* SENSE REVERSAL: Wait for all processors to exit */
  while (entry_count < NUM_PROCS) {}
```

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. The flag can be a sense reversal flag, to prevent intervention of adjacent barrier operations. A serious bottleneck arises with this algorithm, since busy waiting to test the value of the flag occurs on a single, shared location. Moreover, since our lock implementation accesses the semaphore memory through the interconnect, operations that require mutually exclusive access to shared resources introduce additional traffic on the network.

Compiler-generated parallel code may include more barriers than necessary, so it is important to reduce the cost of a single barrier operation to a minimum. To this aim we consider a *Master-Slave barrier* algorithm. In this approach the *controller* is responsible for locking and releasing *workers*. This is accomplished in two steps. In the *Gather* phase, the *controller* waits for *workers* to notify their arrival on the barrier. This operation is executed without resource contention, since every worker signals its status on a separate flag, e.g. separate locations of an array.

```
int gather[NWORKERS];
```

After this notification step, *workers* enter a waiting state, where they poll on a private location.

```
/* Each processor has a private copy of this variable */
int release;
```

In the *Release* phase of the barrier, the *controller* broadcasts a *release* signal on each slave's poll flag.

Each worker notifies its presence on the barrier through the `Worker_Enter` function, by writing at the location corresponding to its *id* in the *gather* array .

```
void Worker_Enter() {
  int ent = release;

  gather[PROC_ID] = 1;
  while (ent == release)
    ;           /* Busy wait on a private variable */
}
```

The value of the *release* flag is read upon entrance, then busy waiting is executed until this value is changed by the *controller*. The *Gather* and *Release* stages are initiated

by the *controller* through the `Controller_Gather` and `Controller_Release` functions shown below.

```
/* Gather workers on the barrier */
void Controller_Gather()
{
  int i;
  for (i=0; i<(NUM_PROCS-1); i++)
    while (!gather[i])
      ;          /* Wait for current worker to arrive */
}

/* Release workers */
void Controller_Release()
{
  for (i=0; i<(NUM_PROCS-1); i++)
  {
    int *rel = <point to i-th processor relase flag> */
    (*rel)++;
  }
}
```

The *Master-Slave barrier* algorithm is expected to eliminate the bottleneck implied by the use of shared counters. Authors of (2) leverage a similar barrier implementation. Anyhow, the traffic generated by polling activity is still injected through the interconnect towards shared memory locations, potentially leading to congestion. This situation may easily arise, for example, when the application shows load imbalance in a parallel region.

To address this issue we divert all the polling traffic towards local memories to every core. This can be done by exploiting a distributed implementation of the barrier algorithm, i.e. allocating each of the slave's poll flag onto their local L1 SPM, and using a message passing-like approach for signaling. In this way, the numer 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.

2.5 Experimental Evaluation

The automatic parallelization framework described in this chapter has been analyzed in details by means of a cycle-accurate virtual platform(3) that models all essential system components. The fundamental parameters for our system are shown in table 2.1.

processor	ARM7, 200Mhz
data cache	4KByte, 4 way set associative latency 1 cycle
instruction cache	8KByte, direct mapped latency 1 cycle
scratchpad memory	16KByte latency 1 cycle
shared memory	latency 2 cycles
AMBA AHB	32 bit, 200Mhz, arbitration 2 cycles

Table 2.1: Architectural components details

2.5.1 Barrier cost

In this section we compare the performance of the three barrier implementations described in Sec. 2.4.3. The first one is a *centralized shared barrier* with sense reversal. The second is a *Master-Slave barrier* with flags allocated in the shared memory. The third is a distributed *Master-Slave barrier* with flags spread among SPMs. The direct comparison of these barriers is shown in Fig. 2.4.

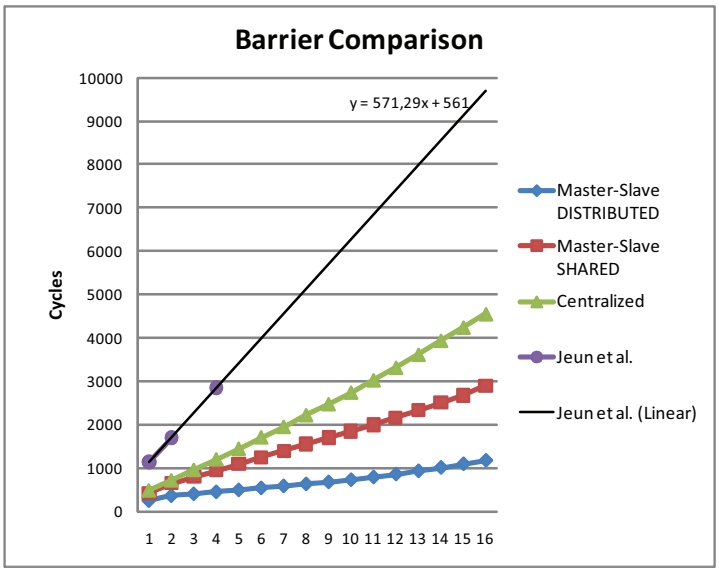


Figure 2.4: Cost of barrier algorithms with increasing number of cores.

The experiments have been carried out by executing barrier code only on the platform. No other form of communication between cores takes place, thus allowing to estimate how the algorithm scales with increasing traffic for synchronization only.

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 we can compute 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 we report here results published by Jeun et al. (4) 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 barrier*, as expected, mitigates the effects of the bottleneck due to contended resources. The cost for each of the phases of the algorithm is plotted in Figure 2.5.

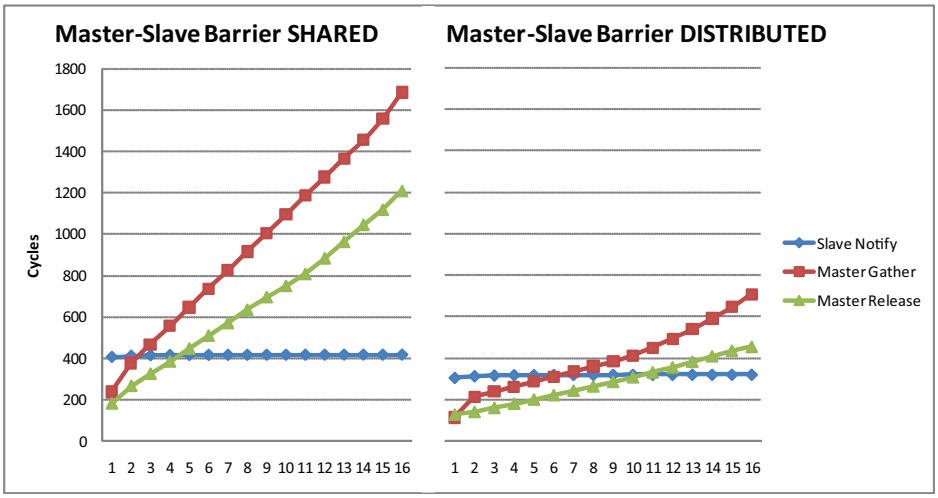


Figure 2.5: Shared and distributed implementations of the Master-Slave barrier.

Even when barrier flags are allocated in the shared memory the cost for synchronizing 16 cores is reduced to $\approx$ 3000 cycles (gather + release). Linear regression indicates a slope of $\approx 150 \times N$. Employing a distributed algorithm allows to completely remove the traffic due to busy-waiting. This ensures the lowest-cost implementation between those envisioned. Synchronization among 16 cores costs around 1100 cycles, with a tendency of $\approx 56 \times N$.

2.5.2 Runtime library performance

To investigate the cost imposed over program execution time by runtime services two synthetic patterns are adopted, to stress representative corner cases in which a program kernel is either *computation-dominated* or *communication-dominated* (i.e. *cpu-* or *memory-bound*). The simulations sweep both in the number of processors and in the size of data. Performance plots show the breakdown of parallel code execution on different processor counts into three main contributions:

- **Init time**: Time required for initialization library routines to complete
- **Synchronization time**: Time spent on barriers
- **Effective execution time**: Time spent over parallel computation

Each of these contributions was also measured in an ideal scenario, and the difference between these numbers and the actual timings collected from the benchmark is referred to as an overhead in the plots. We model the ideal cases as follows.

1. Time spent in barrier routines increases non-linearly with the number of cores because of the increased contention on the shared bus. Since we are only interested in time increase due to synchronization (i.e. polling activity), we force the *controller* to gather *workers* only after they have already entered the barrier. In this case the *controller* has complete ownership of the bus, and only needs to check each flag once.
2. Ideal parallel run-time is estimated by executing on a single processor the share of work it would be assigned if the program was parallelized to evenly divide the workload among n cores. So, for example, if eight processors are to process a vector of 32 elements in parallel, each core would process 4 data elements. The ideal execution time is that a single CPU would take to process a vector of 4 elements.

2.5.3 Communication-Dominated Benchmark

The first set of experiments leverages a square matrix filling kernel, and serves the purpose of investigating how our runtime services and barrier implementation behave in terms of scaling and cost. The impact of the dataset size (i.e. the number of

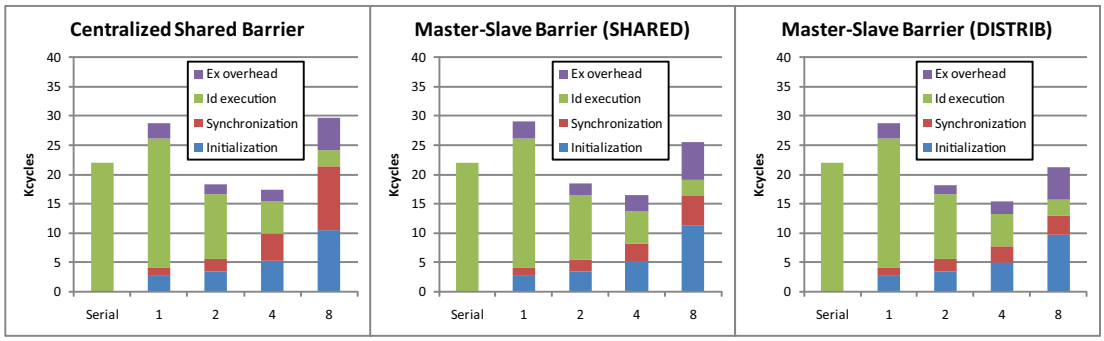


Figure 2.6: Performance results for communication dominated applications (matrix SIZE=32).

matrix rows/columns), has also been taken into account in the experiments. The matrix size is parameterized with a `SIZE` macro. No computation is performed on array elements, so communication towards memory is expected to become the bottleneck limiting parallelization speedup.

```
for(i=SIZE*id/nprocs; i<SIZE*(id+1)/nprocs; i++)
  for(j=0; j<SIZE; j++)
    A[i][j] = 1.0;
```

The plot in Fig. 2.6 shows the results gathered for matrix SIZE = 32. It depicts the breakdown of the different contributions, measured as the overall number of cycles taken by each operation. *Initialization* represents the number of cycles needed for the library initialization routines to complete.

Synchronization cost is split into two contributions: *ideal* synchronization cycles and relative overhead (the difference between measured cycles and ideal cycles). The picture shows how the overhead grows with the number of cores. This is due to the additional bus traffic generated by the cores polling over shared synchronization structures. This kind of overhead is strongly dependent on the balancing of parallel threads. With perfectly balanced workload all threads enter the barrier at almost the same time, thus leading to ideal synchronization time. On the contrary, in the worst case a single thread executes for a longer time than the others, which poll on shared barrier flags thus generating the discussed overhead.

Ideal parallel execution time follows the intuitive trend of almost halving with the doubling of the number of processors, but actual measurements show an execution-time overhead that severely limit the potential speedup.

All of the discussed sources of overhead are tangible with this set of experiments due to the very small dataset size, which requires a very short program execution time. This is an important result to understand how fine the granularity of parallel tasks can be made before the overhead for parallelization support in our library overwhelms the parallel speedups.

For processor counts up to 4 parallel execution scales well even for such very fine granularity. For 8 processors, however, the overhead becomes predominant. Inter-processor communication on our platform travels through a shared bus, which is known to be a non-scalable interconnection medium. For this reason, when the number of cores increases the bus gets congested and the requests are serialized, thus lengthening parallel execution. The library has been designed keeping this limitation in mind. From Fig. 2.6 it is possible to see, for example, that the distributed implementation of the Master-Slave Barrier significantly mitigates the synchronization overhead.

On the other hand, nothing can be done from within the library to mitigate the parallel execution overhead, which depends on the contention for shared data in the program. A more scalable interconnection architecture, such as a crossbar, or a Network on Chip (NoC), would solve this issue, as it will be shown in the following chapters.

Finally, the cost for library initialization increases with the number of cores. However, since initialization only occurs at system startup this cost is fixed, and becomes quickly negligible as the granularity of parallel tasks increases.

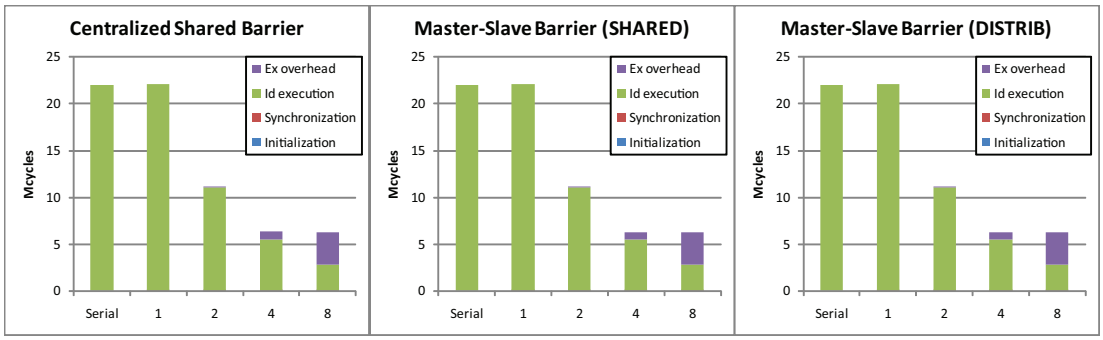


Figure 2.7: Performance results for communication dominated applications (matrix SIZE=1024)

To demonstrate this we repeat the same experiments considering a matrix `SIZE` = 1024. Results for this set of tests are reported in Fig. 2.7. The cost relative to initialization becomes completely negligible, and so does synchronization cost. Parallel

execution time overhead, on the other hand, is still there. As already explained, this depends on the communication-dominated nature of the benchmark and on the limited capabilities of the shared bus in managing the traffic generated by an increasing number of cores.

2.5.4 Computation-Dominated Benchmark

The second set of experiments aims at investigating the cost for library support to parallelization in computation-dominated situations with very few accesses to memory resources. A vector is read in parts from shared memory, which is now declared as cacheable, and a cycle of a variable number of iterations performs several sums on this data. Reducing the accesses to shared memory and spending the largest fraction of parallel time in doing computation should allow linear speedups, as long as the library overheads are not significant. To verify this, we run a set of experiments where both the size of the array and the number of kernel iterations are parameterized – with `SIZE` and `ITER`, respectively.

```
for(i=0; i<ITER; i++)
  for(j=SIZE*id/nprocs; j<SIZE*(id+1)/nprocs; j++)
    tmp += A[j];

return tmp
```

Figure 2.8 shows the results for the execution of this benchmark under varying values of `ITER` and `SIZE`. For tiny array size (32x32) the overheads for synchronization

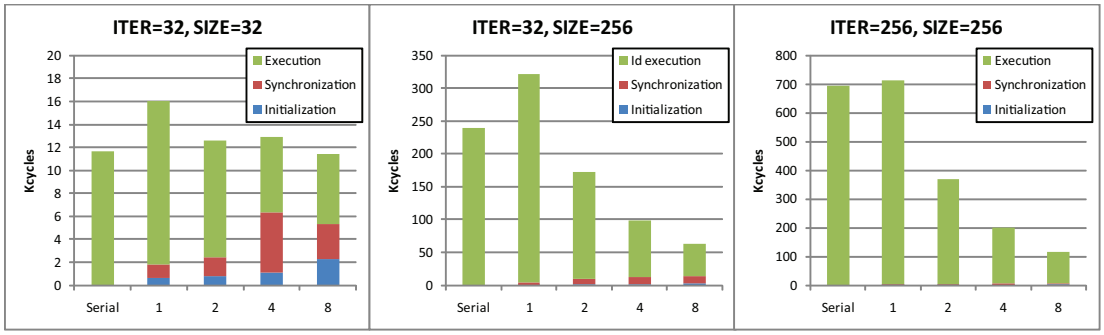


Figure 2.8: Performance results for computation dominated parallel execution.

and initialization are non-negligible when the number of cores increases. Furthermore, a small number of iterations (32) exhibits a lenghtening (w.r.t. serial execution) of

parallel execution time on a single core due to a cold-cache effect. When the array `SIZE` is increased to 256, the library overheads get completely negligible. Similarly, the cold-cache effect disappears when `ITER` is increased to 256. It has to be pointed out that this is very fine data and workload granularity, thus confirming that our library implementation implies only a very little overhead.

2.5.5 JPEG Decoding

In this section we present the results of a real multimedia benchmark: a parallelized version of the JPEG decoding algorithm. After the initialization, performed by every core, the master core starts computing the sequential part of the algorithm (Huffman decoding) while the slaves wait on a barrier. Then the computation is split between cores. Specifically, each CPU executes a *luminance dequantization* and a *inverse DCT* filter over a different slice of the image. Each of these two parallel kernels is synchronized with a barrier. Results are shown in Fig. 2.9. The time taken by the master core

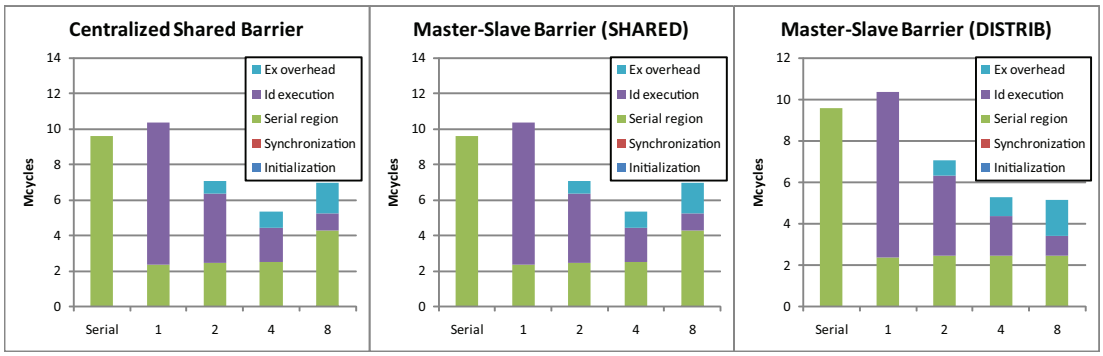


Figure 2.9: Performance results for parallel JPEG decoding

to execute sequential parts of the application is plotted as well. Initialization and synchronization time is negligible due to the overall duration of the program and the small number of barrier invocations. The behavior of the parallel portion of code follows the one we already discussed for the computation-dominated benchmark. During this section of the benchmark the cores access concurrently to shared data[1], thus limiting effective scalability to up to 4 cores. A larger number of processors would perform worse on this shared bus-based architecture. Also, the execution time of the sequential part increases drastically for more than 4 cores. This extra time is due to the polling

[1] As already discussed allowing these data to be cached would reduce this overhead

activity of *workers* on the barrier while the master runs the sequential code. As discussed earlier, this problem can be addressed from the library by adopting a distributed implementation of the *Master-Slave barrier* algorithm.

2.6 Conclusion

The advent of multicore processors in the embedded domain introduced radical changes in software development practices. The many inherent difficulties in the parallelization process call for tools that aid the application developer to accomplish this complex task.

Compiler support to automatic program parallelization is typically focused on the analysis and partitioning of loop nests which do not carry data dependencies. Advanced techniques can also be applied to transform loop nests so as to remove dependencies prior to parallelization.

This kind of parallelism is then mapped onto a SIMD model of computation, where the same loop code is assigned by the compiler to parallel threads which operate on separate portions of the data space. The SIMD execution model requires that threads synchronize on a global barrier after parallel region completion and prior to going ahead in the program. To achieve this goal compilers generate code that invokes the services of a runtime library where all of the support for thread creation, management and synchronization is implemented.

In this chapter we described a support library for the execution of compiler-generated parallel code on an embedded shared memory MPSoC. Global synchronization is achieved by means of lightweight barriers, which implementation has been extensively evaluated using communication and computation intensive synthetic program patterns as well as a real multimedia application.

Results indicate excellent scalability of the library services, whereas a detailed performance analysis shows that the main performance blocker is the bus contention. This interconnection medium limits the applicability of compiler-parallelized programs to up to 8-core architectural templates for CPU-bounded kernels, and only 4-core for memory-bounded kernels. While the results emphasize the importance of low cost barriers with increased number of processors, they also clearly indicate that the library never becomes a bottleneck in parallel execution. This allows to apply compiler-enabled

fine-grained parallel programs on architectural templates where a bigger number of processors is interconnected with a more scalable medium, such as a *crossbar*, or a *Network on Chip*.

In the following chapters we will describe the integration of the library support here presented in a widely adopted parallel programming model for shared memory multiprocessors: OpenMP. The implementation of the compiler and runtime support for this programming framework will specifically target more scalable MPSoC templates.

Bibliography

[1] C. Liu, A. Sivasubramaniam, M. Kandemir, M. J. Irwin, "Exploiting Barriers to Optimize Power Consumption of CMPs", In Proceedings of IPDPS, 2005. 10

[2] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer. Implementing openmp on a high performance embedded multicore mpsoc. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society. 17

[3] MPARM Home Page, *www-micrel.deis.unibo.it/sitonew/mparm.html*. 17

[4] W.-C. Jeun and S. Ha. Effective openmp implementation and translation for multiprocessor system-on-chip without using os. *Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific*, pages 44–49, 23-26 Jan. 2007. 19

[5] Jennifer M. Anderson and Saman P. Amarasinghe and Monica S. Lam, "Data and computation transformations for multiprocessors", In Proceedings of the 5th ACM SIGPLAN symposium on Principles and practice of parallel programming, 1995 10

[6] Jennifer M. Anderson and Monica S. Lam, "Global optimizations for parallelism and locality on scalable parallel machines", In PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation, 1993 10

[7] Mary H. Hall and Saman P. Amarasinghe and Brian R. Murphy and Shih-Wei Liao and Monica S. Lam, "Detecting coarse-grain parallelism using an interprocedural

parallelizing compiler", In Supercomputing '95: Proceedings of the 1995 ACM IEEE conference on Supercomputing (CDROM), 1995 10

[8] Michael E. Wolf and Monica S. Lam, "A data locality optimizing algorithm", In Proceedings of the Conference on Programming Language Design and Implementation, 1991 10

[9] L. Xue and O. Ozturk and F. Li and and I. Kolcu, "Dynamic Partitioning of Processing and Memory Resources in Embedded MPSoC Architectures", In Design Automation and Test in Europe (DATE'06), Munich, Germany, 2006 10

[10] O. Ozturk and G. Chen and M. Kandemir, "A Constraint Network Based Solution to Code Parallelization", In Proc. Design Automation Conference (DAC)[Nominated for Best Paper Award], 2006 10

[11] Jong-eun Lee and Kiyoung Choi and Nikil D. Dutt, "An algorithm for mapping loops onto coarse-grained reconfigurable architectures", In LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language compiler and tool for embedded systems, 2003, pages 183–188 10

[12] J. Sampson, R. Gonzlez, J.F. Collard, N.P. Jouppi, M. Schlansker, "Fast Synchronization for Chip Multiprocessors", In ACM SIGARCH Computer Architecture News, 2005. 11

[13] J. Sampson, R. Gonzlez, J.F. Collard, N.P. Jouppi, M. Schlansker, B. Calder, "Exploiting Fine-Grained Data Parallelism with Chip Multiprocessors and Fast Barriers", In Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture MICRO 39, 2006. 11

[14] J. Li, J.F.Martnez, M.C. Huang, "The Thrifty Barrier: Energy-Aware Synchronization in Shared-Memory Multiprocessors", In Proceedings of the 10th International Symposium on High Performance Computer Architecture HPCA '04 , 2004. 11

[15] S.Kumar, D.Jiang, R.Chandra, J.P.Singh, "Evaluating Synchronization on Shared Address Space Multiprocessors: Methodology and Performance", In Proceedings of the 1999 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, 1999 11

[16] J.M.Mellor-Crummey, M.L.Scott, "Algorithms for Scalable Synchronization on Shared Memory Multiprocessors", In ACM Trans. on Comp. Sys., 1991 11

Chapter 3

Evaluating OpenMP support costs on MPSoCs

The ever-increasing complexity of MPSoCs is making the production of software the critical path in embedded system development. Several programming models and tools have been proposed in the recent past that aim at facilitating application development for embedded MPSoCs. OpenMP is a mature and easy-to-use standard for shared memory programming, which has recently been successfully adopted in embedded MPSoC programming as well. To achieve performance, however, it is necessary that the implementation of OpenMP constructs efficiently exploits the many peculiarities of MPSoC hardware. In this chapter we present an extensive evaluation of the cost associated with supporting OpenMP on such a machine, investigating several implementative variants that efficiently exploit the memory hierarchy. Experimental results on different benchmark applications confirm the effectiveness of the optimizations in terms of performance improvements.

3.1 Introduction

Advances in multicore technology have significantly increased the performance of embedded Multiprocessor Systems-on-Chip (MPSoCs). As more and more hardware functions are integrated on the same device, embedded applications are becoming extremely sophisticated (4) (12). This increased complexity is making the production of software the critical path in embedded system development. Embedded software design for a

multicore platform involves parallel programming for heterogeneous multiprocessors, under performance and power constraints (17). Being able to satisfy such constraints requires programmers to deal with difficult tasks such as application/data partitioning and mapping onto suitable harware resources. Efficiently tailoring an application to the underlying ISA, communication architecture and memory hierarchy is key to achieving performance.

Several programming models and tools have been proposed in the recent past that aim at facilitating application development for embedded MPSoCs. A widely adopted approach is that of extending standard languages from the uniprocessor domain, such as C, with specific constructs to express parallelism. Language-extension approaches require that the programmer provides information on where and how to parallelize a program by means of annotations.

OpenMP (18) is a well-known example of language extension with annotations, which has recently gained much attention in the embedded MPSoC domain. OpenMP is a de-facto standard for shared memory parallel programming. It consists of a set of compiler directives, library routines and environment variables that provide a simple means to specify parallel execution within a sequential code. It was originally designed as a programming paradigm for Symmetric Multi-Processors (SMP), but recently many implementations for embedded MPSoCs have been proposed (8) (16) (10) (13) (14).

There are two main benefits in the OpenMP approach. First it allows programmers to continue using their familiar programming model, to which it adds only a little overhead for the annotations. Second, the OpenMP compiler is relieved from the burden of parallelism extraction and can focus on exploiting the specified parallelism according to the target platform. An OpenMP program is retargetable as long as there exists an associated OpenMP compiler for the target multicore processor. With this respect, since GNU GCC adopted the GOMP OpenMP implementation (1), many GCC-enabled embedded MPSoCs boast an OpenMP translator. However, platform-specific optimization cannot be achieved by a translator only, since OpenMP directives only allow to convey to the compiler high-level information about parallel program semantics. Most of the target hardware specificities 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 design of the runtime library. The original GCC implementation (`libgomp`) 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 layout which features NUMA shared segments and assumes no cache coherence, but is rather explicitly managed by the compiler or the programmer.

We found that an efficient exploitation of the memory hierarchy is key to achieving a scalable implementation of the OpenMP constructs. In particular, 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 wise placement of compiler-generated support metadata
3. Reducing the cost for synchronization directives

In this chapter we present an extensive evaluation of several implementative variants of the necessary support to OpenMP constructs, which take advantage of the peculiarities of the memory hierarchy.

3.2 Background and Related Work

Several OpenMP implementations for MPSoCs have been presented in recent research. One of the most welcome is undoubtedly that for the STI Cell BE (16). While being closely related to ours in the careful exploitation of hardware resources to achieve performance, it presents in practice a completely different set of implementative challenges and choices. The Cell processor is a distributed memory machine, where SPEs can only communicate with each other by means of DMA transfers from/towards the main memory. The abstraction of a shared memory is thus provided by means of a software implementation of a coherent caching mechanism. On the contrary, our architectural template allows every processor to directly access each memory bank in the hierarchy. For this reason, the implementative solutions here presented cannot be directly supported on the Cell memory architecture.

Authors of (10) present an OpenMP implementation for OS-less MPSoCs. They provide optimized implementation of the `barrier` directive, which we use as a direct

term of comparison in Sec. 3.5. In (13) (14) is described an extended OpenMP programming framework for the Cradle 3SoC platform. Custom directives are provided to enable parallel execution on DSPs and to exploit specific banks of the memory hierarchy for data placement. In (8) Chapman et al. describe a set of extensions that could make OpenMP a valuable and productive programming model for embedded systems. They also provide an initial implementation for a TI C64x+ -based MPSoC, with a memory hierarchy very similar to the one we propose.

In the cited papers, however, a detailed evaluation of the implementative solutions they describe is missing. We try to fill this gap by presenting an extensive set of experiments aimed at highlighting the actual costs to support OpenMP programming construts on a representative and generic MPSoC template. We then propose several implementative variants to reduce the cost of most common OpenMP programming patterns.

3.3 Target Architecture

The simplified block diagram of our MPSoC 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 ST STBus protocol, which supports burst interleaving, multiple outstanding and split transactions, thus providing excellent performance and scalability for the considered number of cores. Support for synchronization is provided through a special hardware semaphore device.

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 Fig. 3.2. The shared memory is physically partitioned in several memory segments, each of which may be associated (i.e. tightly coupled, or placed in close spatial proximity) to a specific PE.

Each PE has on-tile L1 memory, which features separate instruction and data cache, plus scratchpad memory (SPM). Local L1 SPMs are tightly coupled to processors, and thus very fast to access. 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 by local L1 cache. Accessing the local L2 memory of a different PE is possible, but requires appropriate cache controls actions to avoid working with a stale copy of data that local processor may have brought in cache.

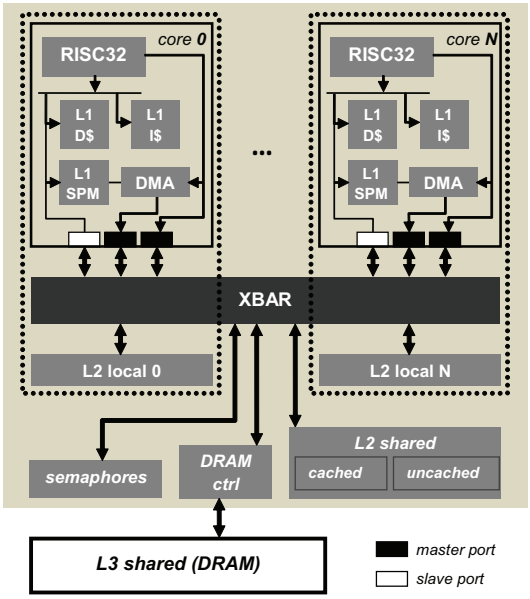


Figure 3.1: Target architectural template.

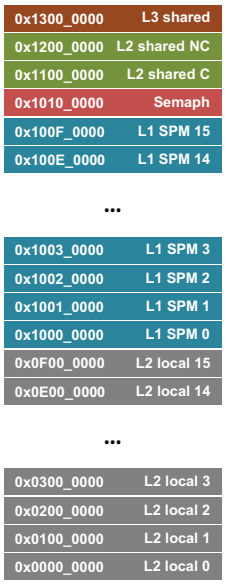


Figure 3.2: PGAS.

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. This requires a cache coherence protocol to be implemented. OpenMP specifies a relaxed consistency memory model, which requires that a coherent view of shared data is enforced only at specific synchronization points. Cache coherence is thus enforced through software flush instructions in our runtime library.

Finally, threads executing on different PEs can also exchange data through the off-chip shared L3 DRAM memory, which is also mapped in the address space of processors.

3.4 OpenMP Support Implementation

An OpenMP implementation consists of a code translator and a runtime support library. The framework presented in this chapter is based on the GCC 4.3.2 compiler, and its OpenMP translator (GOMP). 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 we explain the needed modifications to the compiler and runtime to achieve functionality and performance on the generic MPSoC architectural template presented in Sec. 3.3.

3.4.1 Execution Model

OpenMP adopts the fork-join model of parallel execution. An OpenMP program begins as a single thread of execution, the master thread. The master thread executes sequentially until it encounters a `#pragma omp parallel` directive. Here, a team composed of the master thread and zero or more additional threads executes concurrently a set of implicit tasks defined by the code inside the parallel construct. Beyond the end of the parallel construct the threads synchronize on a barrier, then only the master thread resumes execution. To support this execution model the compiler is in charge of altering the original program flow by replacing a `parallel` block with code that originates multiple dynamic instances of the annotated task. This goal is typically achieved by outlining the code within parallel regions into compiler-generated functions. The GCC GOMP compiler also takes this approach. The parallel block is replaced with calls to the runtime library, where the actual fork/join mechanism is implemented.

The GCC implementation of the runtime library (`libgomp`) is built on top of the Pthreads library. Pthreads require abstraction layers that allow tasks on different cores to communicate. Examples of OSs supporting this feature include SMP Linux (11) and TIs DSP BIOS (2). SMP Linux is suitable for SMP architectures because it provides a shared symmetric view. The heterogeneous nature of MPSoCs, however, is often more suited to AMP (Asymmetric Multi-Processing) programming, where a separate OS is installed on each core and is responsible for handling resources on that core only. Inter-core communication to implement the OpenMP execution model requires specific support, and has significant associated overheads (8).

For this reason, we do not leverage the GCC `libgomp` library, and re-designed the runtime environment from scratch, implementing it as a custom lightweight library where the master core is responsible for orchestrating parallel execution among available processors (similar to (5) (8) (10)). We assume a fixed allocation of the master and slave threads to the processors. At boot time the executable image of the program+library is loaded onto each processor's 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 the master to provide useful work to do. The master core starts execution of the OpenMP application. Since only the master thread executes the sequential parts of the program, when a parallel region is encountered there is the need to notify the slaves about where to find shared code and data. For this reason pointers to the outlined function and to shared data are passed to the runtime environment through the `GOMP_parallel_start` function. Different from the original GOMP compiler, our customized translator only emits a call to this function, where the entire fork/join mechanism takes place.

```
#pragma omp parallel {...}
```

gets transformed into

```
GOMP_parallel_start (foo.omp_fn.0, &mdata);
```

where `foo.omp_fn.0` is a pointer to the outlined function and `mdata` is compiler-generated metadata to support data sharing (see Sec. 3.4.2). In our implementation of this function, the master core copies this information in a predefined location for the slaves to see, then notifies them about the availability of work to do. Master and slave cores then start executing the outlined parallel code. 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 into 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 interferent traffic on the interconnect. To ensure this, we adopt a

message exchange mechanism, 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 the `GOMP_parallel_start` the master sends a message containing task and frame pointers in the queues of all slave cores.

Key to minimizing the overhead associated with the join mechanism is the choice of a lightweight barrier algorithm. We discuss barrier implementation in Section 3.4.3.

3.4.2 Data Sharing and Memory Allocation

OpenMP provides several clauses to specify the sharing attributes of data items in a program, but all of them can be broadly classified into shared and private classes of data, 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 univocal 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. In this way each thread refers to a private copy of the variable. We do not need to modify this behavior with our platform, since private data gets by default allocated onto local L2 memories to each core, thus ensuring the correct semantics for the `private` clause.

Data items annotated with sharing clauses are typically declared within the scope of the function enclosing a parallel directive. This implies that the variable declaration lays within the task which is mapped onto the master thread, and for this reason it is invisible to slaves. Once the program starts, only the master core executes sequential parts, where shared variable declarations belong to. This implies that shared variables reside 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 very common solution to deal with this issue is that of relying on a sort of marshalling operation where the compiler generates metadata containing pointers to shared data. More precisely, 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 address of shared variables into metadata, then passes the structure's address to the runtime environment, which in turn makes it available to 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 Symmetric Multi-Processors (SPM) with Pthreads support this solution provides correct data sharing semantics, since the stack of the master thread is allocated in a memory region which is accessible to slave threads, and performance, since data is accessed through multi-level coherent cache-based memory subsystem.

On architectures with explicitly managed memory performance is trickier to achieve. 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 of the interconnection medium.

We thus consider this default data sharing implementation solution as a baseline for our investigations, to which we will later refer to as **Mode 1**.

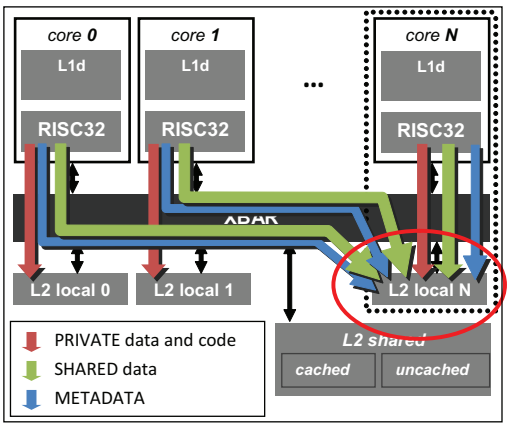


Figure 3.3: Allocation Mode 1.

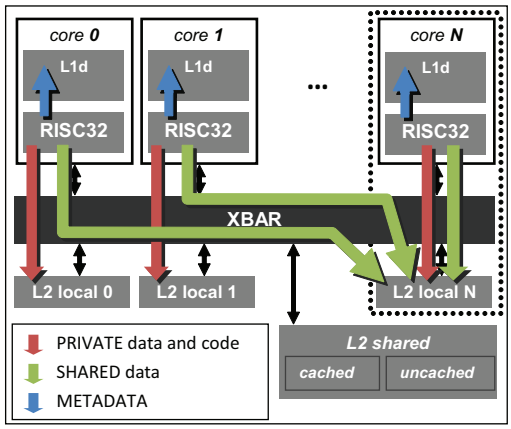


Figure 3.4: Allocation Mode 2.

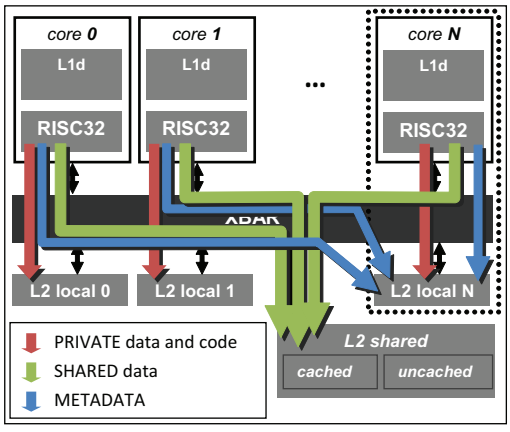


Figure 3.5: Allocation Mode 3.

Here slave processors access shared data/metadata from the master core's local L2 memory. Since this memory bank also hosts all of master processor's private code and data, we expect it to be delayed by other processors' activity on memory, as shown in

Fig. 3.3.

The memory hierarchy of the considered MPSoC template is complex, and physical allocation of shared data and metadata can lead to very different performance results. For this reason we explore a set of compiler-directed placement alternatives 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. Our 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 allows to remove all the traffic towards the master core's L2 local memory due to accesses to metadata (cfr, Fig. 3.4), and will be later referred to as **Mode 2**.

Since most of the 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 out of the master core's local L2 memory. Static declarations of shared data items are first transformed into pointer declarations, and then pointed to the shared L2 memory. Since by default the heap is mapped onto each processor's L2 local memory as well, dynamically allocated variables (i.e. pointers initialized through a `malloc()`) are re-initialized to point to the shared L2 memory. We call this placement scheme **Mode 3**. Similar to **Mode 2**, it aims at reducing the accesses from slave processors to the master's local L2 memory by

entirely shifting the traffic generated by accesses to shared data towards a "dedicated" memory block (see Fig. 3.5). The combination of **Mode 2** and **Mode 3** is called **Mode 4**.

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. Many OpenMP applications exploit data parallelism at the loop level, where shared arrays are accessed by threads in (almost) non-overlapping slices. In this case array partitioning techniques (6) (3) (7), or 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 we allow shared data to be placed on a cacheable region of the shared L2 memory. If metadata resides on the master core's local L2 we call this placement scheme **Mode 5**. If metadata is replicated onto every core's L1 SPM we call it **Mode 6**.

All of the discussed performance issues apply to global data as well. For this reason we customized the GOMP compiler to treat global variables similar to `shared` variables.

3.4.3 Synchronization

OpenMP provides a number of means to perform synchronization between parallel threads, the main directives being `atomic`, `critical` and `barrier`. Groups of instructions that need to execute atomically can be enclosed within a `critical` section, whereas the `atomic` directive allows to mark a single instruction for mutually exclusive execution. The compiler marks the beginning and the end of a critical/atomic section with calls to library functions which acquire/release a lock. We leverage hardware semaphores to implement support for these directives.

Global synchronization can be performed with the `barrier` directive. In the OpenMP programming model barriers are often implied at the end of parallel regions or worksharing directives. For this reason they are likely to overwhelm the benefits of parallelization if they are not carefully designed taking into account hardware peculiarities and potential bottlenecks.

We consider the three barrier implementations described in the previous chapter.

3.5 Experimental Results

In this section we present the experimental setup and the results achieved. An instance of the MPSoC template described in Sec. 3.3 has been implemented within a SystemC full system simulator (15). The effect of all the implementative variants describes in the previous sections has been evaluated on several benchmarks from the *OpenMP Source Code Repository* (9) benchmark suite.

3.5.1 Synchronization

In this section we evaluate the impact of the three barrier implementations described in the previous chapter on most common OpenMP programming patterns. The first one is a centralized shared barrier with sense reversal. The second is a Master-Slave barrier with flags allocated in the shared memory. The third is a distributed Master-Slave barrier with flags spread among L1 SPMs.

To investigate the impact of different barrier algorithms on real program execution we give results for three benchmarks containing representative patterns in OpenMP programs.

1. `#pragma omp single:` When the `single` directive is employed only a thread is active, while the others wait for it to complete execution on the barrier. We model this behaviour with a synthetic benchmark in which every iteration of a parallel loop is only executed by the first encountering thread.

2. **Matrix multiplication:** This benchmark employs a variant 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 very 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. Being the number of inner

iterations not equal (and possibly very different) for every point, parallelizing the outermost loop with static scheduling leads to very unbalanced threads.

Results for each of these benchmarks are shown in Fig. 3.6.

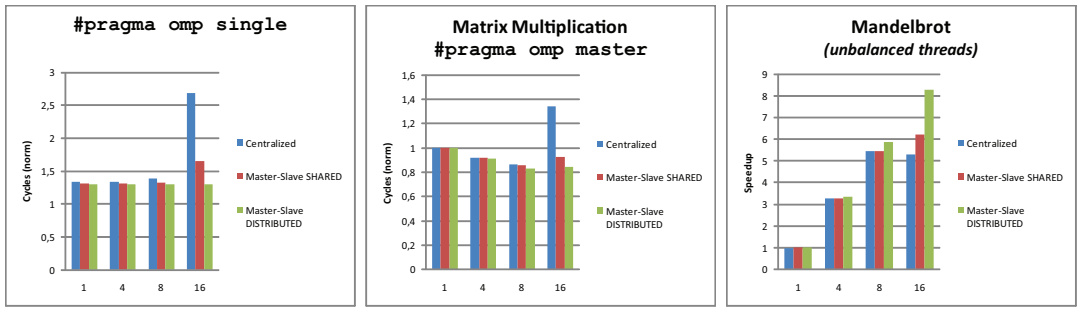


Figure 3.6: Impact of different barrier algorithms on real programs.

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 incresing 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.

3.5.2 Data Sharing and Memory Allocation

In this section we explore the effect of placing shared data and support metadata onto different memory modules in the hierarchy. We provide evidence that 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.

As explained in Section 3.4.2, there are a number of possible allocation combinations that can be explored in our MPSoC. We run our benchmarks under each of the possible placement variants:

- **Mode 1:** The default OpenMP placement. Data and metadata live in the master thread's stack, which physically resides onto master core's local L2 memory segment. Slave cores access them from there. This configuration is considered as a baseline for our experiments.

- **Mode 2:** Shared data still resides onto master core's local L2 memory, where it was originally placed from the compiled program. Metadata, on the contrary, is replicated and transferred onto each core's scratchpad by means of a DMA transfer upon entrance into the parallel region. This is expected to reduce contention on master core's L2 memory.

- **Mode 3:** Shared data is allocated onto the non-cacheable segment of the shared L2 memory. Metadata resides on the master core's local L2 memory. This is expected to significantly reduce contention on master core's L2 memory.

- **Mode 4:** Shared data is allocated onto the non-cacheable segment of the shared L2 memory. Metadata is replicated onto every scratchpad and accessed from there. This configuration reduces to a minimum the number of accesses to the master core's L2 memory for data sharing.

In case a program is memory bounded and most of the accesses are performed towards shared arrays, high-contention on a single memory bank is bound to re-appear. In this situation, as discussed in Section 3.4.2, splitting arrays and allocating each partition on a different memory block allows to mitigate the effect of request serialization on a single memory device port. To investigate the effect of this kind of contention, we consider two additional placement variants which leverage the data cache to implement array partitioning:

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

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

The considered allocation modes are summarized in Table 3.1. We run all of our

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

Table 3.1: Shared data and metadata allocation variants

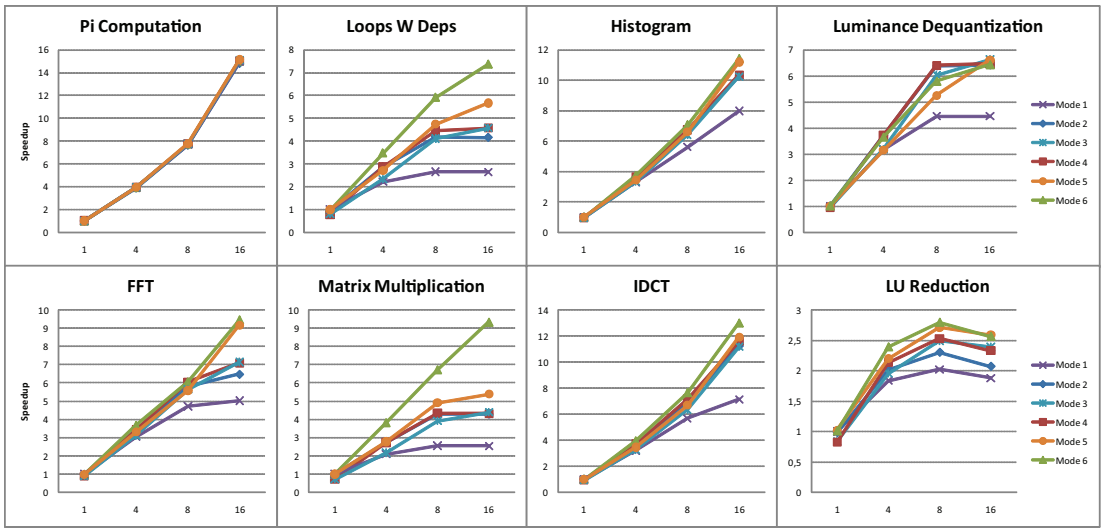


Figure 3.7: Scaling of different allocation strategies for data sharing support structures.

benchmarks under each of the described modes. The barrier adopted for this set of experiments employs the distributed Master-Slave algorithm. Results of this exploration are reported in Figure 3.7.

The curves there plotted show the scaling of the execution time speedup with the number of cores. The speedup is referred 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 w.r.t. default placement **Mode 1**, with the exception of benchmark *Pi Computation*, which shows no difference between modes.

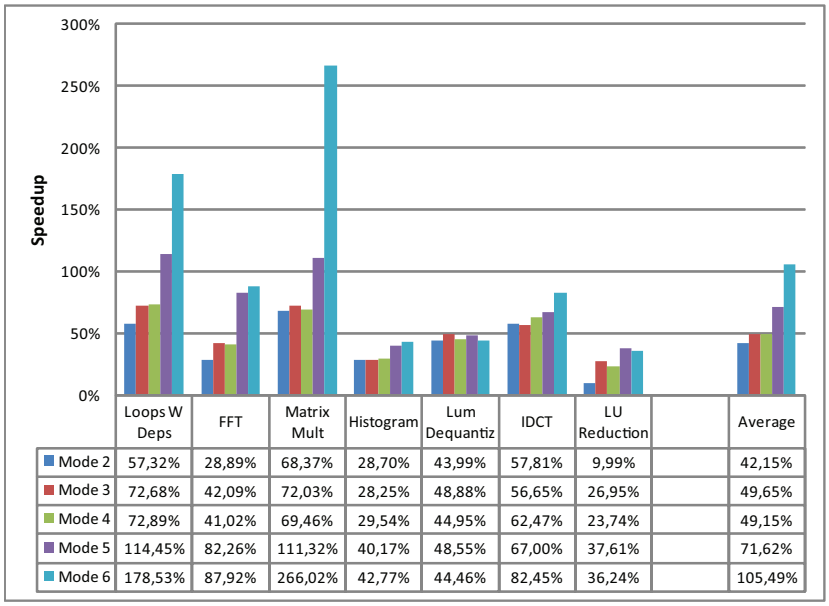
Pi computation computes pi by means of numerical integration. Parallel threads compute a given number of integration steps, then a reduction operation sums all contribution into a shared variable. The reduction operation is implemented in such

a way that all processors accumulate partial results onto a private variable, which is physically mapped onto each core's local L2 memory. No contention arises during this operation, and so neither the network nor any memory port ever gets congested. This placement doesn't change for any of the allocation modes. The shared variable is only accessed at the end of the parallel loop. Every processor atomically updates it by adding its partial integration result. Since the update only happens once, and has a very brief duration w.r.t. loop execution, changing the allocation of the shared variable does not result in any advantage, and thus all modes deliver similar performance.

Focusing on the rest of the benchmarks, it can be seen that replicating metadata onto every processor's L1 SPM (**Mode 2**) allows significant improvements with any number of processors. For processor counts up to 8 is on average faster than simply allocating shared data onto 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 our interconnect medium is congested when both metadata and data are accessed from master core's local L2 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 gets stuck and performs identical to modes 2 and 3. This happens in particular for benchmarks *Loops W Deps*, *Luminance Dequantization* and *Matrix Multiplication*. Figure 3.8 shows the speedup of **Modes 2-6** against **Mode 1** for 16 cores only. This plot shows that on average **Mode 2** allows $\approx 42\%$ speedup. **Mode 3** achieves $\approx 50\%$ speedup, but **Mode** 4 can not do any better. This behaviour is due the 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 master core's 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. It has to be explained that this happens 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



	Loops W Deps	FFT	Matrix Mult	Histogram	Lum Dequantiz	IDCT	LU Reduction		Average
Mode 2	57,32%	28,89%	68,37%	28,70%	43,99%	57,81%	9,99%		42,15%
Mode 3	72,68%	42,09%	72,03%	28,25%	48,88%	56,65%	26,95%		49,65%
Mode 4	72,89%	41,02%	69,46%	29,54%	44,95%	62,47%	23,74%		49,15%
Mode 5	114,45%	82,26%	111,32%	40,17%	48,55%	67,00%	37,61%		71,62%
Mode 6	178,53%	87,92%	266,02%	42,77%	44,46%	82,45%	36,24%		105,49%

Figure 3.8: Speedup of several data sharing support variants against the baseline for 16 cores.

matrix rows. Row elements are operated on in parallel within the innermost loop. Since the number of row elements gets 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 a bigger number of available core, thus justifying the performace degradation from 8 to 16 cores.

3.6 Conclusion

Software development in the embedded MPSoC domain is becoming increasingly complex as more and more feature-rich hardware is being designed. OpenMP is a mature standard for shared memory parallel programming. Even if it was designed more than a decade ago for Symmetric Multi-Processors (SMP), its adoption in the embedded MPSoC domain has recently been proposed by several researchers. Pioneers pointed out the challenges in porting OpenMP to complex and heterogeneous MPSoCs. Even if there is consensus on the fact that OpenMP may be both suitable and profitable for MPSoC programming, it is also unanimously recognized that compiler and run-

time support must be revisited to account for the peculiarities of embedded MPSoC hardware.

All of the 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. We rather target a generic and representative embedded MPSoC template, and describe an OpenMP implementation based on a modified GCC 4.3.2 compiler and on a custom runtime library. We also present an exhaustive study of the performance achieved by several implementative variants of the necessary support for OpenMP constructs. We demonstrate that careful implementation of the shared memory abstraction on top of non-uniform and explicitly managed memory hierarchies is key to achieving performance.

Bibliography

[1] Gomp - an openmp implementation for gcc. [Online]. Available: http://gcc.gnu.org/projects/gomp/. 32

[2] Ti dsp/bios. [Online]. Available: http://focus.ti.com/docs/toolsw/folders/print/dspbios.html. 36

[3] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, and C. D. Offner. Extending openmp for numa machines. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 48, Washington, DC, USA, 2000. IEEE Computer Society. 42

[4] G. Blake, R. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, November 2009. 31

[5] C. Brunschen and M. Brorsson. Odinmp/ccp - a portable implementation of openmp for c. *Concurrency - Practice and Experience*, 12(12):1193–1203, 2000. 37

[6] R. Chandra, D.-K. Chen, R. Cox, D. E. Maydan, N. Nedeljkovic, and J. M. Anderson. Data distribution support on distributed shared memory multiprocessors. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 334–345, New York, NY, USA, 1997. ACM. 42

[7] B. Chapman, F. Bregier, A. Patil, and A. Prabhakar. Achieving performance under openmp on ccnuma and software distributed shared memory systems. *Concurrency and Computation: Practice and Experience*, 14(8-9):713–739, 2002. 42

[8] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer. Implementing openmp on a high performance embedded multicore mpsoc. In *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society. 32, 34, 36, 37

[9] C. d. S. F. Dorta, A.J. Rodriguez. The openmp source code repository. *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*, pages 244– 250, 9-11 Feb. 2005. 43

[10] W.-C. Jeun and S. Ha. Effective openmp implementation and translation for multiprocessor system-on-chip without using os. *Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific*, pages 44–49, 23-26 Jan. 2007. 32, 33, 37

[11] M. T. Jones. Linux and symmetric multiprocessing: Unblocking the power of linux smp systems, 2007. [Online]. Available: http://www.ibm.com/developerworks/library/l-linux-smp/. 36

[12] L. Karam, I. AlKamal, A. Gatherer, G. Frantz, D. Anderson, and B. Evans. Trends in multicore dsp platforms. *Signal Processing Magazine, IEEE*, 26(6):38–49, November 2009. 31

[13] F. Liu and V. Chaudhary. Extending openmp for heterogeneous chip multiprocessors. *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 161–168, 6-9 Oct. 2003. 32, 34

[14] F. Liu and V. Chaudhary. A practical openmp compiler for system on chips. *International Workshop on OpenMP Applications and Tools, WOMPAT 2003*, pages 54–68, June 2003. 32, 34

[15] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon. Analyzing on-chip communication in a mpsoc environment. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 20752, Washington, DC, USA, 2004. IEEE Computer Society. 43

[16] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang. Supporting openmp on cell. In *IWOMP '07: Proceedings of the 3rd international workshop on OpenMP*, pages 65–76, Berlin, Heidelberg, 2008. Springer-Verlag. 32, 33

[17] H. woo Park, H. Oh, and S. Ha. Multiprocessor soc design methods and tools. *Signal Processing Magazine, IEEE*, 26(6):72–79, November 2009. 32

[18] www.openmp.org. Openmp application program interface v.3.0. 32

Chapter 4

Data Partitioning for Distributed Shared Memory MPSoCs

Most of today's state-of-the-art processors for mobile and embedded systems feature on-chip scratchpad memories (SPM). To efficiently exploit the advantages of low-latency high-bandwidth memory modules in the hierarchy there is the need for programming models and/or language features that expose such architectural details. On the other hand, effectively exploiting the limited on-chip memory space requires the programmer to devise an efficient partitioning and distributed placement of shared data at the application level.

In this chapter we describe the necessary language features in embedded MPSoC parallel programming to efficiently exploit explicitly managed memories. Such features provide a means for the programmer to convey architectural awareness to the compiler, which can optimize program execution time and/or energy consumption. The programming framework here described combines the ease of use of OpenMP with simple yet powerful language extensions to trigger array data partitioning. The compiler and runtime environment exploit profiled information on array access count to automatically generate data allocation schemes optimized for locality of references.

4.1 Introduction

The scaling limitations of uniprocessors have led to an industry-wide turn towards chip multiprocessor (CMP) systems. CMPs are becoming ubiquitous in all computing

domains, from high performance supercomputers to embedded systems (1). Focusing on the latter, today's embedded multi-processor systems on a chip (MPSoC) are capable of executing sophisticated tasks such as audio and video decoding. This requires a design that is capable of delivering high performance, while fitting tight energy and area budgets (2).

Memory system design is a critical problem for MPSoC platforms. The chip has a limited amount of on-chip memory that must be augmented by bulk commodity memory off-chip. On-chip memory is faster and consumes less power but is of limited capacity (3) (4). Most of todays state-of-the-art processors for mobile and embedded systems feature on-chip cache and/or scratchpad memories (SPM) (5) (6) (7) (8) (9) (10). Caches and SPMs are both made of SRAM cells. Caches are composed of tag and data RAM plus management logic that makes them mostly transparent to the software. On the contrary, SPM consists of a simple array of SRAM cells, without a tag RAM and complex comparator logic. This simpler design has several practical advantages, particularly profitable in the embedded domain. SPM requires up to 40% less energy and 34% less area than cache (11). Additionally, SPM cost is lower and its software management makes it more predictable, which is an important feature for real-time systems. Typically the SPM is mapped into the physical address space as a contiguous block of fast memory. Unlike caches, it is the up to the programmer (possibly with the help of the compiler) to determine what parts of the code/data are placed in the SPM. Placing the most frequently accessed parts of the program into the SPM can reduce both the energy consumption and the execution time of an application (12).

Obviously, the choice of a memory model is closely coupled with the choice of a parallel programming model which allows to efficiently map an application on top of the hardware resources (13) (14). Typically, the higher degree of architectural awareness is exposed to the programming model, the more its ease of use is affected. Parallel extensions of traditional programming languages such as C, are appealing because they adopt a simple and intuitive memory model, to which programmers are accustomed to, with a single address space. Providing the abstraction of a shared memory and hiding the details of the memory hierarchy organization increases ease of use, but may fail to map efficiently to these architectures. On the other hand, exposing the memory hierarchy organization to the application puts on the programmer the burden of efficiently

partitioning data and distributing it through explicit transfer onto appropriate memory banks.

This scenario calls for programming models and tools that ease this process. In this chapter we describe a parallel programming framework for embedded MPSoCs based on OpenMP (15). The OpenMP standard is very mature, but since it was originally intended for homogeneous cache-coherent SMPs, it lacks any architectural awareness. We combine the ease of use of the OpenMP programming style with the efficient exploitation of the memory hierarchy by extending the API with custom data placement features. Specifically, we provide directives and clauses that allow the programmer to mark arrays for partitioning and distributed allocation across the memory hierarchy. Exploiting profiled information on array access count, during parallel regions array partitions (tiles) are allocated onto the SPM local to the processor that accesses it most frequently. The programmer can also outline custom program regions where to capture access locality. This information is then stored in specific metadata hold in the runtime environment. Compiler-instrumented array references inspect metadata to determine the position of the target array partition in memory. Array data layout in memory is automatically updated through DMA movements upon region enter and exit.

4.2 Background and Related Work

Embedded parallel applications from the image processing domain are typically based on decomposition of data array processing across parallel threads (SPMD). Image arrays are logically divided in blocks or slices, each of which is independently processed. Efficiently mapping this computational model onto a machine with multiple NUMA memory modules requires data (array) structures to be partitioned in smaller chunks (often called *tiles*), each of which has to be placed close to the processor that most frequently references it. In the recent past, a plethora of programming models and APIs providing features to ease this task has seen the light (see Sec. 4.2.1). To achieve performance, typically these solutions require a high degree of programmer involvement, both in identifying efficient partitioning schemes and in modeling communication through appropriate memory modules.

We developed such techniques as a set of extensions of the OpenMP API, within an MPSoC-suitable implementation of the OpenMP compiler and runtime library (see

Chapter 3 or (16)). Due to its well-know ease of use, the OpenMP standard has already saw several (mainly vendor-specific) implementations with customized features for NUMA machines. Related work on array data partitioning and OpenMP extensions for data distribution on NUMA machines is discussed in Sec. 4.2.2. Typically, NUMA OpenMP APIs expose directives for data distribution and loop parallelization with affinity scheduling (i.e. loop iterations accessing a given array tile are assigned to the processor in closest spatial proximity with the memory bank hosting the tile), but the programmer is in charge of properly using the directives. This requires deep understanding of the memory access pattern of the application. To mitigate this problem, we extend our programming framework with tools that automatically determine an efficient data placement, based on application profiling. There is a vast amount of literature describing static and dynamic techniques to place array data onto a single SPM (in a single-processor system). Our target MPSoC, however, is a multicore system where each processing element (PE) has fast access to a local SPM bank, and incurs NUMA latency when accessing remote SPM banks. Only recently research has started focusing on allocation techniques on multiple SPMs. We describe related work with this respect in Sec. 4.2.3.

4.2.1 Programming the Memory Hierarchy

Programming models and compilers to explicitly take into account the memory hierarchy at the application level have been actively investigated in the recent past by several researchers (see (17) (13) (14) for good overviews). Sequoia (18) is a programming language that abstractly exposes hierarchical memory in the programming model and provides language mechanisms to describe communication vertically through the machine and to localize computation to particular memory locations within it. This execution model is particularly well-suited to data parallel computations. It enforces strict locality of computation, since tasks run in isolation on a processor and can only access data from within local memories. On the other hand, inter-node communication is much more complicated and much less performance-efficient, since it has to take place through dedicated sub-tasks. This, in turn, makes it very difficult to model different kind of parallelism (e.g. task parallelism) with Sequoia constructs. Our approach allows much more flexibility in modeling different kinds of parallelism and the cost for inter-processor communication is carefully optimized.

Recently, general purpose computation on graphic processors has received a lot of attention as it delivers high performance computing at rather low power. CUDA (9), Compute Unified Device Architecture, proposed by the GPU vendor NVIDIA, is a programming model for General Purpose Graphics Processing Units (GPGPU) computing. It provides a multi-threaded Single Instruction Multiple Data (SIMD) model for implementing general-purpose computations on GPUs. Although the unified processor model in CUDA abstracts underlying GPU architectures for better programmability, its memory model is exposed to programmers. The main difficulty in writing CUDA programs is that the programmer is deeply involved in leveraging hardware features to achieve performance. As an example, loop tiling and memory coalescing, necessary to achieve performance (and program correctness) are entirely left to the programmer. Tools like CUDA-lite (19) are being proposed to take from the programmer the burden of doing manually the transformations to exploit local memories. It has to be pointed out, however, that CUDA-lite performs no automatic optimization, but rather relies upon information from the programmer provided via annotations to perform its transformations. Our programming framework efficiently couples the ease of programming with annotations with profile-based compiler optimizations and runtime support for effective exploitation of the memory hierarchy.

4.2.2 Data Partitioning and OpenMP Extensions for NUMA architectures

Data partitioning has been widely studied in the context of NUMA multiprocessors and distributed shared memory systems (20). In particular there is a vast amount of literature dealing with the integration of such techniques in OpenMP (21) (22) (23). All these works introduce OpenMP API extensions to enable distribution of shared arrays over multiple memories in a NUMA architecture, and they are closely related to ours 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 have far-reaching consequences. Traditional NUMA machines were organized as clusters of computing nodes (e.g. the SGI Origin), where inter-node communication has orders-of-magnitude lower speed than local operations. Remote memory access took place under

software abstractions such as virtual memory paging. In these systems the cost associated to such a memory management layer is hidden behind the huge communication cost. In embedded MPSoCs the scenario is completely different. First, paged virtual memory is not supported in hardware as it would be way too expensive to replicate a complex MMU for all the element of a large-scale on-chip data-processor array[1]. Second, all communication travels on-chip, where latency is much lower and bandwidth is much higher. These major differences lead to completely different set of implemention choices, that we will describe in details in the following sections. Moreover, all the cited work provides means to convey to the compiler information to trigger and direct data partitioning and distributed placement, but all decisions about how to actually carry out those tasks are completely left to the programmer. We provide similar facilities, but in addition to that we enrich our compilation toolchain with features to automatically devise an efficient placement, leaving to the programmer only the burden of expressing its will to enable locality optimizations on a particular data structure.

We are not the first to consider OpenMP in the context of MPSoC programming. Even though OpenMP was originally intended for symmetric multiprocessors (SMP) with shared address space, its appealing ease of use has led to the recent development of several MPSoC-specific implementations, one representative example being O'Brien et al.'s port for the STI Cell BE processor (25). Anyhow, being the Cell a distributed memory machine, where accelerators can only reference data from local memories, the abstraction of a shared memory is provided through a compiler-controlled software cache that initiates DMA operations whenever needed. Implementing the OpenMP memory model on top of the Cell memory model has the advantage of making memory management completely transparent to the programmer, but comes at the cost of maintaining coherency in software. In our MPSoC processors can access local memories to other cores, but at an increased cost. As such, we face a different problem of optimizing array tiles placement through a physically distributed shared memory space.

[1] in MPSoCs organized as a battery of DSP/accelerators coordinated by a general-purpose control processor virtual memory may be supported on the latter, which usually runs a complex OS and performs high-level orchestration (24)

4.2.3 Scratchpad Management

Methods for static and dynamic placement of data and program objects on SPM have been proposed in a huge amount of related work in literature. Representative examples can be found in (26) (12) (27) (28) (29) (30) (31) (32). However, while constituting a fundamental background to our work, the approaches here described focus on data/code allocation on a single SPM (considering single-core architectures) We try instead to solve the problem of efficiently allocating array data over multiple SPMs, with NUMA organization.

The role of (multiple) scratchpads in the multicore domain has only recently come to the research forefront. In (33) Yanamandra et al. explore the benefits of replacing the cache with the SPM at different levels of the memory hierarchy. They also consider hybrid architectural templates, similar to ours, in which both SPM and cache are considered. The focus of their work, however, is on evaluating the role of SPMs in optimizing sparse matrix-vector multiplication kernels, and thus their approach is less general than ours.

Abdelkhalek and Abdelrahman introduce in (34) program constructs and runtime support to dynamically manage data stored in the SPMs. Their approach is very similar to ours, in that they provide the programmer with compiler directives aimed at directing data allocation to specific memory banks (i.e. SPMs) and setting an affinity between tasks and SPMs where data has been allocated. Anyhow, the proposed techniques are less powerful than ours in two respects. First, data allocation is entirely left to the programmer, which must necessarily have insights of the application to efficiently use the proposed directives. On the contrary, we also provide a profile-based automatic placement technique which requires much less programmer effort. Second, only entire data structures can be placed on SPMs, thus encountering scalability bottlenecks when multiple cores are accessing the same data item. We efficiently solve this issue by providing support for (array) data partitioning.

In (35) Kandemir et al. present a compiler-directed optimization strategy for exploiting multiple SPMs (the Virtually-Shared Scratch-Pad Memory) in an embedded multiprocessor system. Their approach is oriented toward eliminating extra off-chip DRAM accesses caused by data sharing, and relies on performing loop transformations to simplify the reuse pattern or to improve data locality. The main limitation of the

approach is that it only works with statically analyzable (i.e. at compile time) applications with regular access patterns. Furthermore, the size of the SPM must be known at compile time. Our techniques do not require this information, which can be retrieved at runtime, and is capable of dealing with irregular applications as well. Finally, our automatic allocation techniques take into account the NUMA organization of typical VS-SPM systems.

4.3 Target Architecture

In the followin of this chapter we consider two architectural templates, whose simplified block diagrams are shown in figure 4.1. The platform consists of a configurable number of processing elements(PEs), based on a simplified (RISC-32) design without hardware memory management (i.e. no MMU is available). The interconnection network is a cross-bar, based on the ST STBus protocol, which supports burst interleaving, multiple outstanding and split transactions, thus providing excellent performance. Support for synchronization is provided through a special hardware semaphore device, implemented as a set of memory-mapped on-chip registers with *test-and-set* semantics. Inter-core communication takes place from within the external (DRAM) shared memory, moreover, each processor tile hosts a small amount of low-latency, high-bandwidth SRAM. We consider two alternative implementations for the on-chip memory hierarchy.

In the first template each PE features private data and instruction caches. In order to maintain architectural scalability, hardware cache-coherency is not supported. A consistent view of the shared memory from concurrent multiprocessor accesses is enforced through explicit insertion of cache flush operations in software. In this template on-chip memory management is accomplished entirely in hardware, and thus is used as a term of comparison for our techniques, which explicitly manage SPMs.

In the second architectural template on-tile SRAM is implemented as a combination of per-core small unified cache plus scratchpad memory (SPM). Overall SPM space is organized as a Partitioned Global Address Space (PGAS) system, a.k.a. Virtually Shared Scratch Pad Memory (VS-SPM). In this memory model each PE has fast local access to its associated SPM (typically 1 cycle), since communication towards this memory exploits a dedicated connection, and does not travel across the system interconnect. Processors can also access remote SPMs through a dedicated slave port,

but incur a non-uniform memory access (NUMA) latency (1 order of magnitude slower than local references). Accessing the off-chip main memory is 2 orders of magnitude slower than accessing the local SPM. In our framework SPM space is entirely devoted to hosting shared array data[1]. Private data and code to each thread, as well as scalar variables – currently not managed – are thus accessed through the cache.

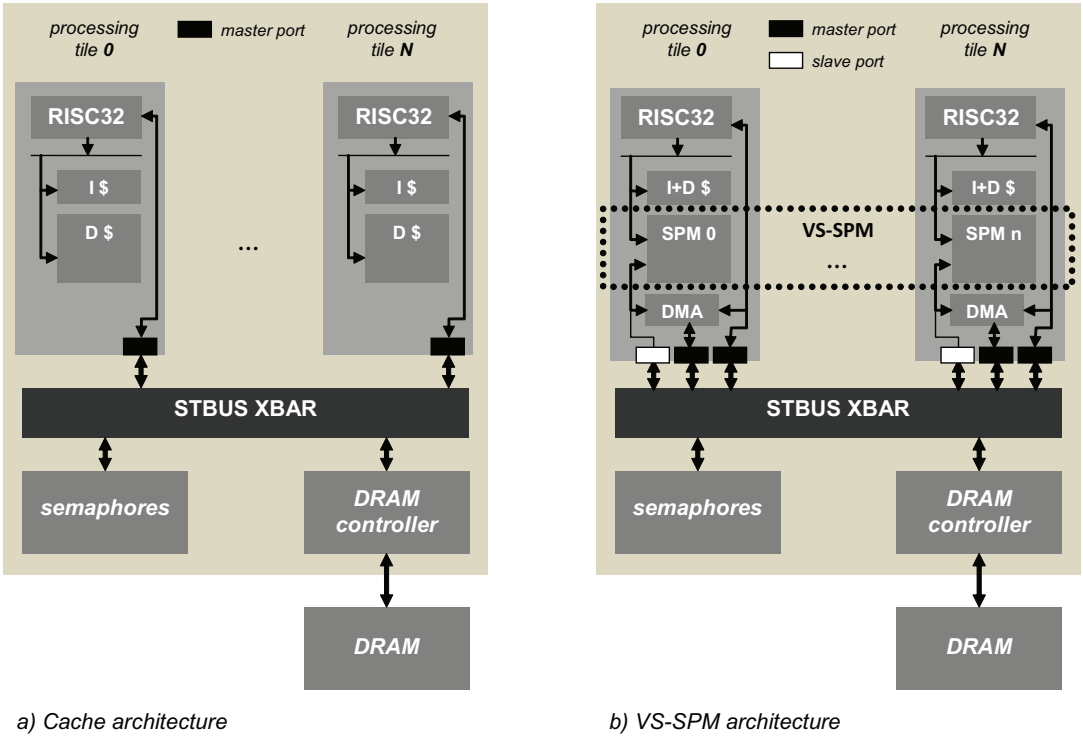


Figure 4.1: Architectural templates.

To reduce the overhead for data movement we also couple each processing element with dedicated transfer hardware, i.e. a Direct Memory Access (DMA) engine. The DMA enables memory transfers between the SPM and the main memory without processor involvement. It is composed by a controller, which acts as the main interface with the processor, and a transfer engine. The processor programs DMA jobs and checks their status by writing/reading into/from the controllers address space. DMA jobs contain information on the source/destination address and stride, which can be conveniently set/reset by leveraging the methods of a small API for DMA programming(36).

[1]As discussed in the previous chapter, a very small amount of each SPM's space is used for the implementation of runtime environment services, such as synchronization, task allocation, etc.

Initiating a DMA transfers comes at the cost of invoking these functions to write necessary information onto the DMA controller registers. Whenever the transfer engine is free and the job queue is not empty, the DMA controller initiates a transaction on the transfer engine. The transfer engine generates the necessary burst accesses to the bus and SPM. The burst size to the bus can be configured through a parameter. The transfer engine contains a 64B queue to store incoming/outgoing data waiting to be forwarded to the SPM or to the main memory. Finally, it is connected to the memories outside the processing tile through a master port. Communication towards the on-tile SPM exploits a dedicated connection.

4.4 An Extended OpenMP API for Efficient SPM Management

OpenMP provides a relaxed-consistency shared-memory model, which specifies that each thread is allowed to have its own temporary view of the memory. This can represent any kind of intervening structure, such as machine registers, cache, or other local storage, between the thread and the memory. The temporary view of memory allows the thread to cache variables and thereby avoid going to memory for every reference to a variable. A coherent view of the memory is implicitly enforced when entering and leaving a parallel region, at other synchronization points, and – explicitly – through the use of the `flush` directive.

We can tailor the implementation of such a memory model to our architecture, taking into careful account the peculiarities of the NUMA organization of the shared memory hierarchy. In the following sections we describe how the original programming interface has been augmented with several features to efficiently manage the NUMA shared memory, providing the programmer with easy-to-use yet powerful means to express the need for locality optimization on candidate data structures.

4.4.1 OpenMP Extensions for Array Partitioning

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

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

When declaring an array as distributed in a program, an unique identifier (hereafter called DISTVAR_ID) is annotated into the intermediate representation (IR) for that variable declaration. The *tilesize* parameter is used to specify the granularity of partitioning, namely the size – expressed in terms of array elements – of the elementary tile. If no such parameter is given, the array lays un-partitioned in memory, but unlike statically declared data, it can be assigned to different memory modules in the hierarchy if profitable.

In general, when an array is partitioned, each tile may reside on a different memory module. Partitioning techniques have been implemented in the past, on distributed shared memory machines, by relying on hardware and OS support for virtual memory, and – in particular – page migration facilities. On our MPSoC there are no such facilities available, so there is the need to design an efficient and lightweight software mechanism to correctly locate tiles in memory. Since our platform features a Partitioned Global Address Space (PGAS) organization of the shared memory hierarchy, this can be done without emulating heavy-weight page migration abstractions. Still, partitioning an array introduces addressing difficulties. Since data tiles can become not-contiguous in physical memory, we can no longer reference the data structure as a whole by simple offset computation. Addressing an element involves finding its physical address, specified by a memory bank number and an offset within that bank. We propose further extensions to the OpenMP API to deal with this issue.

Concurrent accesses to arrays take place within parallel regions, so there is the need for language features to specify array access pattern at a specific region. To this aim, the custom clauses `tiled` and `split` can be coupled to the `parallel` directive, or to the worksharing `for` and `sections` directives, to capture two different access pattern to `distributed` arrays in the program.

1. Each processor may access more than one array partition. In this case, at each reference there is the need to identify which partition is being accessed, and at which offset. In this case the array is annotated with the `tiled` clause.

2. Each processor accesses a single array partition. If this can be ensured by the programmer, there is no need to check which partition is being accessed at every array reference (this is only computed at first access). In this case the array is annotated with the `split` clause.

Annotating an array with the `split` or `tiled` clause within a parallel region replaces the standard OpenMP `shared` qualifier for that array within the region. The semantics of the `shared` clause guarantee that every reference to a shared array from every thread in a parallel region points to the same shared location. The `split` and `tiled` clauses augment this behavior by providing each thread with the knowledge of the base address of each array partition. This information can be retrieved by inspecting the compiler-generated metadata array `tiles[DISTVAR_ID][TILE_ID]`. The first index of the `tiles` array uniquely identifies a `distributed` array in the program, whereas the second index identifies a specific partition (tile) of the array. Dereferencing metadata through a couple (DISTVAR_ID, TILE_ID) allows retrieving the base address of a specific array tile. For each `tiled` array access the compiler identifies a TILE_ID by dividing the offset of the reference by the size of a tile. This information was conveyed to the compiler through the `distributed` directive, along with the DISTVAR_ID.

The physical placement of array tiles in memory (i.e. the addresses stored within metadata) can be either directed by the programmer, or decided based on profile information. We describe the generation of data layouts in Section 4.4.4. The instrumentation process triggered by the use of the two custom clauses is shown in Figure 4.2. On the topmost side we show a code snippet of a histogram creation kernel with two `distributed` arrays. The parallelization scheme employed in this example leverages static scheduling (i.e. loop iterations are evenly divided among threads in contiguous, equally-sized chunks), so the image array `img` is accessed by threads in separate, non-overlapping slices of size TSIZE2. In this case we can partion the array into tiles that perfectly match the threads footprint, and annotate it for `split` access in the loop. On the contrary, the array `hist` is accessed with an irregular pattern, being it subscripted by array `img`. In this case it is impossible to determine an optimal partitioning for the array `hist` a-priori. Here we partition it in blocks of size TSIZE1, generating as many tiles as processors[1]. Since all processors could reference every tile, the array is annotated for `tiled` access in the loop. On the bottom side of the figure we show the transformed code of the outlined parallel region (encircled by a red frame). Here is possible to notice the difference between `split` and `tiled` accesses. When annotating

[1]In general, the finer the partitioning, the more references can be satisfied from local memory in a profile-based allocation strategy. On the other hand, the footprint of metadata grows with the number of array tiles, so a tread-off has to be found.

Figure 4.2: Compiler instrumentation of `tiled` and `split` arrays

an array for `split` access, the programmer ensures that all array references fall within a single tile, so the base address for that tile is retrieved from metadata only once at the beginning of the region and stored in a local pointer. Every array reference in the thread is then resolved through this pointer. All accesses to tiled arrays, on the contrary, are instrumented with code for tile identification and metadata lookup. It is intuitive that accesses to `split` arrays are less costly than `tiled` accesses, since in the latter case we re-compute the address at every reference.

4.4.2 OpenMP Extensions for Data Movement

Having data layouts change at different regions in the program implies the necessity for data movements. We allow the programmer to trigger different types of transfers, coupled with the use of the partitioning clauses presented in the previous section. Let us consider the slightly modified histogram creation example in Fig. 4.3. Both the arrays `hist` and `img` have been partitioned in as many tiles as processors. Upon entrance into the parallel region, there is the need to copy data into the SPMs. As already pointed out, tiles of the `hist` array are accessed by multiple processors, without information about which tile is being accessed at any istant in time. For this reason it is necessary to

copy the entire array before executing code in the parallel region. Similarly, since array hist is used after the parallel region, it is flushed upon region exit from SPMs back to main memory. To copy in/out entire `distributed` arrays we provide the `copyarrayin` and `copyarrayout` clauses.

```
double A[100];
#pragma omp distributed (A[, tilesize])
...
#pragma omp parallel tiled(A) \
            copyarrayin(A) copyarrayout(A)
{
   Code block
}
```

The compiler defers actual DMA programming to the runtime, into which inserts calls employing the library builtins

```
__builtin_GOMP_copy_to_scratch
              (uint src_addr, uint tsize);
__builtin_GOMP_copy_to_extmem
              (uint src_addr, uint tsize);
```

The semantics of these functions is blocking. DMA transfers are scheduled in a *First Come First Served* fashion to available processors until there are tiles to copy. At the end all processors synchronize on a barrier before going ahead. The parameters passed to the runtime library are the base address of the array in external memory, and the size of a tile. The destination address in SPM space for every tile is automatically retrieved by looking up in the `tiles` metadata array described in Sec. 4.4.1. Once the address is known, the runtime checks wheter the current tile was actually selected for SPM allocation at compile time before actually initiating the DMA transfer. Indeed, due to SPM space limitations or low access frequency, the allocation pass (cfr. Sec.4.4.4) may have decided to leave the tile on the main memory, and access it from there. In this way, DMA movements are strictly limited to those tiles which are guaranteed to achieve some benefits from on-chip allocation. Moreover, since the runtime checks at every tile movement request for available SPM space, it is actually possible to override compile-time decisions on tile allocation, thus making it not necessary to know the actual SPM size at compile time.

The `img` array, on the contrary, has `split` semantics, so every thread accesses a single tile. For this reason, every thread is responsible for copying the target tile onto a local

```
#define ROWS      240
#define COLS      160
#define NPROCS    4
#define TSIZE1    (256/NPROCS)
#define TSIZE2    (ROWS*COLS/NPROCS)

void foo (...)
{
  int hist[256];
  #pragma omp distributed(A,TSIZE1)
  int img[ROWS][COLS];
  #pragma omp distributed(img,COLS)

  #pragma omp parallel                \
              tiled(hist)             \
              copyarrayin(hist)
  #pragma omp for                     \
      schedule(dynamic,1,inout(img))
  for (i=0; i<ROWS; i++)
    for (j=0; j<COLS; j++)
    {
      hist[img[i][j]]++;
    }

}
```

Figure 4.3: Usage of array copy in/out clauses

```
#define ROWS      240
#define COLS      160
#define NPROCS    4
#define TSIZE1    (256/NPROCS)
#define TSIZE2    (ROWS*COLS/NPROCS)

void foo (...)
{
  int hist[256];
  #pragma omp distributed(A,TSIZE1)
  int img[ROWS][COLS];
  #pragma omp distributed(img,TSIZE2)
  int i, j;
  int pid = omp_get_thread_num();

  #pragma omp parallel for            \
              tiled(hist)             \
              split(img)              \
              copytilein(img,pid)     \
              copyarrayin(hist)       \
              copyarrayout(hist)
  for (i=0; i<ROWS; i++)
    for (j=0; j<COLS; j++)
    {
      hist[img[i][j]]++;
    }

  printf("hist[127]=%d", hist[127]);
}
```

Figure 4.4: Usage of tile copy in/out clauses

buffer in its SPM. The programmer can inform the compiler about single tile transfers through the use of the custom `copytilein` and `copytileout` clauses To trigger DMA transfers, the compiler relies to the custom library builtins:

```
__builtin_GOMP_single_copy_to_scratch
              (uint src_addr, uint dst_addr, uint tsize);
__builtin_GOMP_single_copy_to_extmem
              (uint src_addr, uint dst_addr, uint tsize);
```

There is no synchronization among threads when using these clauses, since each processor is only forced to wait for its local transfer to complete before executing parallel code. The `copytilein` and `copytileout` clauses can also be coupled with the `for` construct, to exploit finer partitioning schemes.

`split` access semantics typically takes place at regular loops. Let us consider the slightly different parallelization scheme of the histogram creation kernel in Fig. 4.4. Here dynamic scheduling is employed, so each iteration of the outermost loop is

scheduled independently to first available thread. In this example, this leads to parallel threads accessing the `img` array in contiguous chunks of COLS elements. Again, the partitioning strategy can capture this access pattern by appropriate use of the `distributed` directive

```
int img[ROWS][COLS];
#pragma omp distributed(img,COLS)
```

While for the `hist` array we still need a `tiled` access, and thus a monolitic copyin/-copyout of the entire data structure, the `img` array is accessed with `split` pattern, and thus tiles of the size of an image row can be continuously copied in and out for at the beginning and end of each loop iteration. Furthermore, the parallel loop iterator can be used to identify the tile being processed. This is a very common case in OpenMP programs, and suggests that scheduling clauses for parallel loops could be extended to automatically initiate DMA transfers before executing code in the loop body and after computation has been done. We allow this to be done by providing the `inout` keywork as an optional additional parameter to the OpenMP scheduling clauses `dynamic` and `static`.

```
#pragma omp for schedule(dynamic[, 1, inout(img)])
```

Accesses to the `img` array are treated as `split` references, but they no longer rely on compiler-generated metadata for allocation, since the thread-to-tile access pattern is completely known at compile time. DMA transfers towards thread-private buffers in local SPMs are automatically scheduled by the compiler at the beginning and end of each loop body, exploiting the loop iterator as a tile identifier.

4.4.3 Customizing Program Regions

Since the array access pattern may change at different program regions (e.g. across different parallel loops), the most appropriate data (tile) layout in memory should ideally change accordingly. Stated another way, we are able to deliver best locality of accesses if several instances of metadata arrays are provided, to capture the access pattern occurring at different program points. By default, the lexical extent of the directive to which the `tiled` clause is coupled (i.e. `parallel`, `for`, `sections`) identifies such regions, for each of which metadata describing the array tiles layout in memory is

created. Anyhow, such a granularity may turn out to bee too coarse (or fine) to deliver efficient execution, due to the cost for frequent data movements. Indeed, metadata provides threads with a consistent view of distributed data layout in memory, but there is the need for DMA transfers to actually update the content of SPMs upon entrance into the program region described by metadata itself. A typical example of such a case is when a parallel region is nested within one or more loops (see the LU decomposition benchmark in Sec. 4.5).

```
for (i=0; i<N; i++)
  #pragma omp parallel for tiled(A) \
              copyarrayin(A) copyarrayout(A)
  for (j=0; j<M; j++)
  {
     Code block
  }
```

Changing data layout at every parallel region, in this case, may leads to a big number of DMA transfers, which may overwhelm the benefits of improved access locality.

Depending on architecture-specific cost for DMA on one hand, and remote SPM access on the other, it may be advantageous to sacrifice a little on the side of the access locality to reduce DMA cost. We provide the programmer with the `profiled_locality` directive to outline custom program regions which identify suitable points for data layout re-organization. Accesses to distributed arrays within `profiled_locality` blocks are considered in isolation during the allocation step, and specific metadata for these regions is generated. All the clauses for partitioning and DMA described in Sections 4.4.1 and 4.4.2 can be coupled to the `profiled_locality` directive.

```
#pragma omp profiled_locality tiled(A) \
            copyarrayin(A) copyarrayout(A)
{
  for (i=0; i<N; i++)
    #pragma omp parallel for
    for (j=0; j<M; j++)
    {
       Code block
    }
}
```

4.4.4 Automatic Generation of Data Layouts

As explained in the previous sections, the compiler handles a `tiled` clause by re-computing the address of a distributed array reference within a parallel region. This computation is based on the identification of the tile being accessed and lookup in metadata to retrieve its base address. There may be different ways to determine the content of metadata, which corresponds to finding an efficient allocation for a given partitioning. In the following subsections we describe the two means available in our framework for specifying data placement, namely block/cyclic and profile-based allocation.

4.4.4.1 Cyclic Tile Allocation

Previous work related to ours on data partitioning for NUMA machines completely relies on the programmer to specify a BLOCK or CYCLIC allocation. We provide this option as well. By default, once a partitioning (i.e. a tile/block size) has been specified through the use of the `distributed` directive, tiles are assigned cyclically to SPMs until all tiles have been allocated, or there is no space left on SPMs. A few examples of cyclic allocation are shown in Fig. 4.5

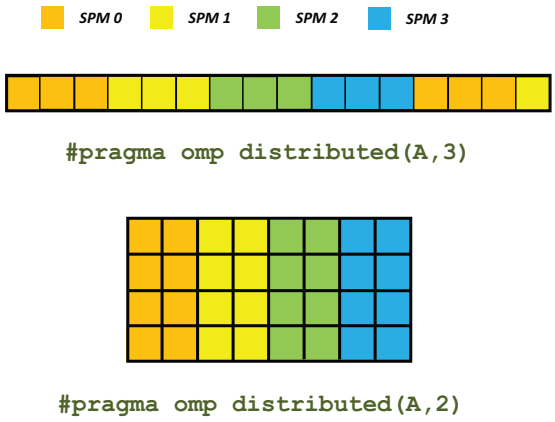


Figure 4.5: Block/Cyclic tile allocation

Cyclic placement works fine with regular applications, where most of the accesses to distributed arrays can be marked as `split` references, or nearest-neighbor computations, where only a few elements are known to reside on remote SPMs. Anyhow, when dealing with less regular applications cyclic allocation may lead to SPM space wastage and poor locality. A better solution to deal with similar scenarios is that of relying on

profiled array access count information. We explain how this option is supported in our framework in the following section.

4.4.4.2 Profile-Based Tile Allocation

By providing a custom `-fomp-distributed` flag, our compiler is capable of instrumenting the program to produce access count information for arrays declared within the `distributed` directive. More specifically, when the programmer is unsure about the access pattern performed at different regions in the program, he can simply annotate possible candidate arrays with the `#pragma omp distributed` directive, compile the program with the `-fomp-distributed` flag, and launch a profile run. During this program run, an execution trace containing all references to distributed arrays from every processor is collected. Every row in the trace file contains an address belonging to a distributed array element and the ID of the processor that performed the access. When this information is available, our toolchain can exploit it to carry out a more efficient allocation strategy, aimed at maximizing the number of local references for every processor. The problem of allocating on SPM space the subset (i.e. a number of tiles) of a given number of partitioned arrays which brings the maximum benefit can be modeled as a variant of the multiple knapsack problem (37). A first constraint in our problem is the size of the global SPM space (i.e. the sum of single SPM sizes), which is often smaller than the array footprints (i.e. the sum of the array sizes). The overall access count to a tile could be a good metric to identify candidate tiles to allocate on-chip. On the other hand, the NUMA organization of the SPM space imposes another constraint. Unless a tile is accessed by a single thread, the choice of a target SPM for placement influences the global cost of all accesses from multiple threads. To minimize this cost we should ideally map the tile onto the SPM local to the processor with the highest access frequency, but in case there is no space left on that SPM things get more complicated.

To solve the allocation problem we adopt an algorithm which implements a greedy heuristic for the knapsack problem (38).

A single access to a distributed array access is represented within the allocation pass through the `mem_access` data structure.

```
/* A memory access descriptor */
typedef struct
{
  /* The ID of the processor accessing the array */
  int pid;
  /* The address of the mem reference */
  int address;
} mem_access;
```

The trace file collected during the profile run is parsed into an array `R` of `mem_access` descriptors, which constitutes the input of the allocation algorithm. The number of available processors P is also passed as an input. Since each processor has an associated SPM, the number of SPMs S is equal to P. The output of the algorithm is metadata describing the target memory bank chosen for every single array tile. The analysis is carried on by exploiting specific `tile` descriptors, whose internal representation within the allocation pass is provided below.

```
/* A tile descriptor */
typedef struct
{
  /* The unique ID of the DISTRIBUTED array to
   * which the tile represented by this struct
   * belongs
   */
  int distributed_array_id;
  /* The ID of the tile within the array */
  int tile_ID;
  /* The ID of the tile w.r.t. global ordering */
  int global_tile_ID;

  /* Base address of the array in memory */
  int dram_address;
  /* Size of the tile (bytes) */
  int tile_size_B;
  /* Base address of the current tile */
  int start;
  /* End address of the current tile */
  int end;

  /* Per-processor tile access count */
  int access_count[P];
  /* Accesses from all processors */
  int overall_access_count;

  /* Cost for allocation on different SPMs */
  double cost[S];
  /* Profit of having the tile on-chip */
  double avg_profit;

  /* Target SPM for allocation */
  int spm_id;
  /* Offset w.r.t. SPM base address */
  int spm_offset;
} tile;
```

Each input memory reference in `R` is inspected in turn to estabilish which distributed array – and which tile within that array – it belongs to (*line 5*). The base address of the distributed array is stored in the `dram_address` field of the target tile descriptor. Along with the size in bytes of a tile – determined through the programmer-specified partitioning granularity and the array type size, and stored in the `tile_size_B` field – this information is necessary to determine the range of addresses belonging to the considered tile, so that every memory reference parsed from the trace file can be brought back to a specific tile. A profit metric to estabilish the benefit obtained by placing a tile on-chip (whatever SPM) is the total number of accesses performed from whatever

Algorithm 1 PROFILE-BASED TILE ALLOCATION(R)

Require: R - A set R[] of mem_access references to distributed arrays within a region.
Ensure: M - Metadata for array tile placement is SPMs.

1: **for all** $mem\_accessR[I] \in R$ **do**
2: Identify which tile J the reference $R[I].address$ belongs to
3: Increase global access counter for tile J
4: Increase processor's $R[i].pid$ access counter for tile J
5: **end for**
6: **for all** tiles $T[I] \in T$ **do**
7: **for all** processors $P[J] \in P$ **do**
8: Compute cost to allocate tile $T[I]$ on SPM $P[J]$
9: **end for**
10: Compute profit for allocating tile $T[I]$ on-chip
11: **end for**
12: **Sort** T by decreasing avg_profit
13: **for all** tiles $T[I] \in$ sorted T **do**
14: **if** $T[I].avg\_profit \leq$ MIN_PROFIT **then**
15: **Continue**
16: **end if**
17: **Sort** array of SPM allocation cost $T[I].cost$ by increasing values
18: **for all** SPMs $J \in$ sorted $T[I].cost$ **do**
19: **if** SPM J has room to allocate tile $T[I]$ **then**
20: **Allocate** $T[I]$ on SPM J
21: **Break**
22: **end if**
23: **end for**
24: **end for**
25: **Generate** M

processor to that tile, so once the tile has been identified, its `overall_access_count` counter is incremented (*line 6*).

On our architecture, accessing a remote SPM encounters a much higher latency than accessing the local SPM. For this reason, to achieve maximum benefit from tile allocation onto SPM space we must be able to satisfy the maximum number of each processor's references from its local SPM. To annotate access frequency information

on a processor basis, each tile descriptor contains an `access_count` array field, which holds a separate counter for each processor (*line 7*).

Based on this information, every SPM in turn is considered for tile placement, and an associated cost is computed. The array field `cost` of the tile descriptor stores the estimated cost for tile placement on different SPMs (*lines 8–10*).

Once the entire trace file has been parsed, all of the program's memory references to distributed arrays are represented with a list of tile descriptors, each of which can be easily accessed through its program-wise identifier (the `global_tile_ID` field in the descriptor), or by specifying the identifier of the distributed array it belongs to (`distributed_array_ID`) and a tile identifier within that array (`tile_ID`).

Since different partitioning granularity can be specified for different distributed arrays (i.e. tile sizes may differ from an array to another), the actual profit of placing a specific tile on-chip is computed by dividing the tile access count by its size, and stored in the `avg_profit` field of its descriptor (*line 11*).

Elements of the tile descriptors array are sorted by decreasing `avg_profit` (*line 12*), so that ordered scanning of the array considers most-advantageous tiles first for SPM placement. For each tile descriptor elements of the `cost` array are sorted by increasing values (*line 16*), then considered in this order (minimum cost first) for SPM placement (*line 17*). If there is no space left on an SPM, the ordered `cost` array is scanned to find the first available SPM slot (*lines 18–20*). The hosting SPM id is stored in the `spm_id` field of the descriptor, and the offset relative to the position of the tile within that SPM is annotated in the `spm_offset` field. If no SPM can host a tile, it is assigned to its original position in external memory. The process continues until all tiles are allocated.

Based on the `spm_id` and `spm_offset` fields of the descriptor the allocation pass generates metadata arrays for `tiled` accesses in the program (*line 21*).

4.4.5 Tool Implementation

Figure 4.6 depicts the entire transformation flow and the tools that we developed. The original application is annotated with custom directives and clauses by the programmer. In particular, the granularity of array partitioning is specified through the `distributed` directive. The rest of the compilation, profiling and optimization process is automated by several scripts, which are globally managed from a top-level `exec_all` program. The entire process can be summarized in the following steps:

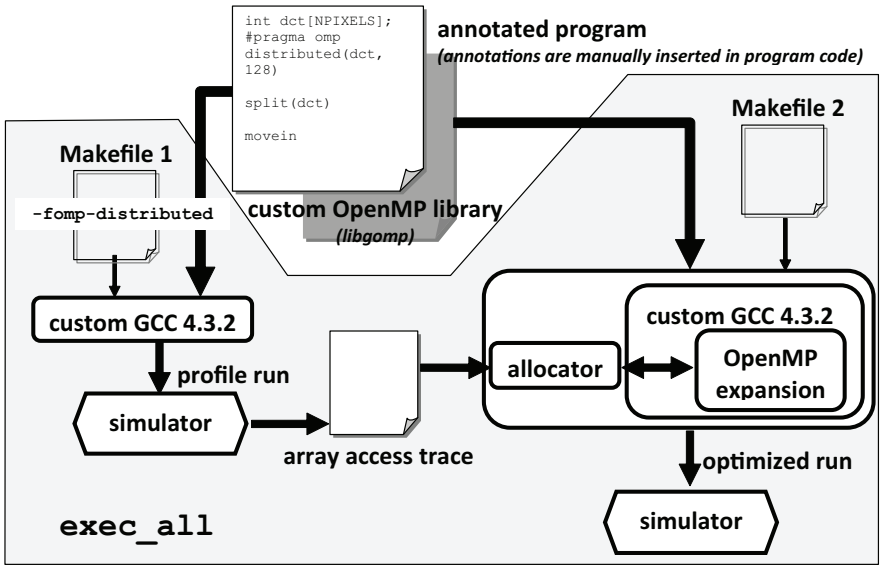


Figure 4.6: Tool flow

1. A first Makefile is generated. Programmer hints about data distribution are discarded, and entire `distributed` arrays are placed in the shared memory.

2. A `-fomp-distributed` flag is passed to the compiler to instrument the application so that accesses to these arrays are monitored.

3. A first simulation step takes place. A script collects array access information into a convenient trace file.

4. A second Makefile is generated which actually triggers array partitioning in the compiler. This task is accomplished by generating look-up operations into allocation metadata.

5. The trace file is fed to our allocation algorithm – which is hooked to the OpenMP expansion pass in the compiler – to generate metadata.

6. A second simulation step takes place, with arrays partitioned and distributed across SPMs.

4.5 Experimental Results

We describe in this section the experimental setup used to evaluate our programming framework and the results obtained.

Architecture Simulation: We implemented an instance of the two platform templates presented in Sec. 4.3 within a SystemC full system simulator (39). Our CMP die hosts 8 processor tiles, based on a RISC32 CPU. The parameters for the on-tile SRAM organization are shown in table 4.1 for the cache-based (architecture A) and SPM-based (architecture B) platforms. Since the focus of this work is on data placement, in both templates code management is entirely accomplished through instruction caches (8KB). Architecture A features a 16KB data cache to cope with data management. The runtime library enforces a consistent view of shared data through flush instructions. Architecture B leverages a 16KB scratchpad memory (SPM) for shared array data, whereas thread-private data is accessed through a small data cache (4KB).

	Architecture A *Cache-based*	*Architecture B* *SPM-based*
I-cache	8KB, direct mapped latency (cycles): 1	8KB, direct mapped latency (cycles): 1
D-cache	16KB, 4way set-assoc latency (cycles): 1	4KB, 4way set-assoc latency (cycles): 1
SPM	–	16KB latency (cycles): 1 (local), 10 (remote)

Table 4.1: On-tile SRAM memory organization

Accesses to local caches and SPMs exploit a dedicated connection and are subjected to only 1 cycle latency. For remote SPMs this cost depends on the internal memory interface latency ($\approx$ 2 cycles), the contention level on the network, the remote memory interface latency ($\approx$ 2 cycles), and the remote SPM latency (1 cycle). We model the zero-load latency for each core to traverse the interconnect for remote memory access with 10 cycles. The cost for an off-chip shared memory access is 100 cycles. If interconnect resources are shared with other concurrent transactions, the latency increases.

Architecture B exploits a DMA engine on each processor tile to reduce the copy cost between main (off-chip DRAM) shared memory and SPMs. DMA transfers are

initiated within our runtime library by means of a small high-level API (36). Initiating/terminating a DMA transfer on the processor has a cost of ≈ 400 cycles (included the call overhead to API functions). Data is transferred in bursts of 8 words.

Benchmarks: We show results obtained with two code kernels, extracted from the *OpenMP Source Code Repository* (40) benchmark suite, and two real applications. All of the considered benchmarks are representative of the memory access patterns found in typical embedded applications from the matrix and image processing (array-intensive) domain:

- **LU decomposition** – This code kernel decomposes a square matrix into the product of a lower triangular matrix (L) and an upper triangular matrix (U). LU decomposition is frequently adopted for matrix inversion, which plays a significant role in 3D graphics
- **FFT** – A Fast Fourier Transform computation kernel based on the Cooley-Turkey algorithm
- **NCC (Normalized Cut Clustering)** – This application is adopted in environmental monitoring through wireless sensor networks. Changes in a monitored area (e.g. people entering a room) are detected by means of a clustering algorithm that computes image affinity between frequently captured images and the reference background (e.g. the image of the empty room)
- **JPEG decoding** – A complete JPEG decoder application

The baseline for all our experiments is a program configuration in which all shared data is placed in the off-chip shared memory, and is accessed from there by means of single transfers. Benchmarks are then executed on architectures A and B, under several data placement variants:

- *CACHE* - Architecture A: Shared data is fetched from main memory through the caches. Coherence is maintained in software (flushes).
- *tiled* - Architecture B: Arrays are annotated with the `tiled` clause and the content of SPMs is updated at each parallel region with the `copyarrayin` and `copyarrayout` clauses.

- *profiled_locality* - Architecture B: The programmer defines custom regions for profile-based data placement by using the `profiled_locality` directive. Data is copied in/out SPMs only once at region enter and exit.

- *inout* - Architecture B: The `schedule` clause for loop parallelization is combined with the `inout` clause. Each thread operates on a local buffer where data is transferred automatically in/out at every scheduling event.

Our plots show the execution time of each benchmark under the above described data placement configurations, normalized to the baseline. Thus, we show both the effectiveness of our techniques in a (data) cache-less machine and a comparison against a cache-only solution. We provide a breakdown of the execution time into three main contributions:

- **CPU+Mem** - Time taken from actual parallel computation, plus the time spent on memory accesses

- **DMA tran** - Time spent on DMA transfers

- **DMA prog** - Time spent on DMA programming

4.5.1 LU Reduction

4.5.1.1 Parallelization

We provide a detailed description of different parallelization schemes on this benchmark to show a concrete usage of our custom directives. Let us consider the LU decomposition code kernel shown in Fig. 4.7. Computation takes place within a triply-nested loop, and is done on progressively smaller submatrices in the lower right-hand corner of the arrays L and M. The outermost loop scans elements on the diagonal, the loop in the middle scans matrix rows, and the innermost loop scans columns. The loop nest in the middle is parallelized with dynamic scheduling, thus originating threads working independently on separate matrix rows. We consider three data distribution variants for this kernel. The simplest one, useful in absence of any insights on the application array access pattern, is that of accessing the arrays with `tiled` references, and updating the content of SPM space at every parallel region. In case the programmer

```
double M[SIZE][SIZE];
#pragma omp distributed(M,SIZE)
double L[SIZE][SIZE];
#pragma omp distributed(L,SIZE)

/* Scan sub-matrices */
for (k=0; k<SIZE-1; k++)

#pragma omp parallel for schedule(dynamic) \
                      tilein(M,k)          \
                      tileinout(M,i)       \
                      tileinout(L,i)
  for (i=k+1; i<SIZE; i++)
  {
    L[i][k] = M[i][k] / M[k][k];
    for (j=k+1; j<SIZE; j++)
    {
      M[i][j] -= L[i][k] * M[k][j];
    }
  }
```

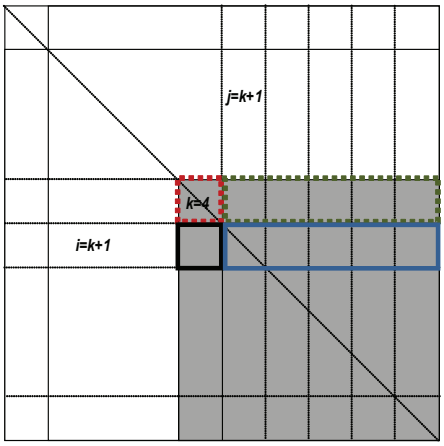


Figure 4.7: LU decomposition kernel

can not determine a suitable partitioning, it is possible to provide tentative values to the `distributed` directive (TSIZE parameter).

The main drawback of this solution is that it implies a high amount of DMA traffic, since the parallel region is nested within a loop, thus requiring frequent re-organization of data layouts in memory. Even if profile based allocation guarantees excellent locality of references within the specified region, in this situation, local accesses are not likely to repay the cost for frequent data movement.

In this case we could consider a coarser program region by enclosing the entire outermost loop within a `profiled_locality` directive, thus triggering a single copy-in and a single copy-out DMA transfer. Considering profile data for a coarser program region is likely to lead to a higher number of non-local references, but the overhead for degraded locality would probably be much smaller than that required for intensive DMA.

```
double L[SIZE][SIZE];
#pragma omp distributed(L,TSIZE);
double M[SIZE][SIZE];
#pragma omp distributed(M,TSIZE);

...
#pragma omp profiled_affinity \
            copyarrayin(M,L) copyarrayout(M,L)
{
  // Outermost loop is enclosed within the
  // PROFILED_AFFINITY directive
  for (k=0; k<SIZE-1; k++)
  {
#pragma omp parallel tiled(M,L)
    {
#pragma omp for schedule(dynamic)
      for (i=k+1; i<SIZE; i++)
      {  Loop body  }
    }
  }
}
```

Finally, it is possible to notice that employing dynamic scheduling (with chunk size = 1) in this program generates threads operating on an entire matrix row. Coupling the `inout` keyword with dynamic scheduling leads to a partitioning with maximum affinity with the thread footprint. Single rows can be brought inside local SPM by each thread before processing, and stored back in their original position after that. Besides updating elements from the i-th row of matrices L and M, each thread also accesses row k from matrix M. Since references to this tile are read-only (dashed line rectangles in Fig. 4.7), it is possible to copy it on each SPM without incurring in memory inconsistency issues. We do this through the use of the `tilein` clause. Since this tile is never written within the loop, there is no need for a `tileout` clause.

```
double L[SIZE][SIZE];
#pragma omp distributed(L,SIZE);
double M[SIZE][SIZE];
#pragma omp distributed(M,SIZE);

...
  for (k=0; k<SIZE-1; k++)
  {
  #pragma omp parallel tilein(M,k)
    {
      #pragma omp for schedule(dynamic,1,inout(M,L))
      for (i=k+1; i<SIZE; i++)
      {  Loop body  }
    }
  }
}
```

4.5.1.2 Results

We execute the LU Reduction benchmark employing all of the three data distribution schemes described above. Results are shown in Fig. 4.8. We consider three partitioning granularities, namely tile sizes equal to 4, 2 and 1 row. Focusing on the three *tiled* bars, as expected a huge penalty for frequent DMA is paid, thus leading to worse performance than the cache. Besides the cost for DMA transfers, a significant amount of time is spent on initiating and terminating the transfers themselves. It is possible to see that this cost increases as the tilesize is reduced, since this implies initiating a higher number of transfers. On the other hand, employing finer partitioning granularity provides better locality, and thus reduces the time spent on memory.

The `profiled_locality` solution allows to remove the DMA overhead, while preserving the benefits of improved locality with finer partitioning. An average 2,3× speedup is achieved with this solution against the cache.

Finally, dynamic scheduling + `inout` DMA provides excellent results, with a 1,8× improvement w.r.t. the cache.

4.5.2 Fast Fourier Transform (FFT)

This program adopts a Cooley-Turkey algorithm for FFT computation. Even in this case the main parallel loop is nested within an outer loop. As confirmed by the results in Fig. 4.9, this leads to the already discussed problems with repeated DMA. Nonetheless, it is possible to notice that for this benchmark this is not a relevant issue. Since parallel

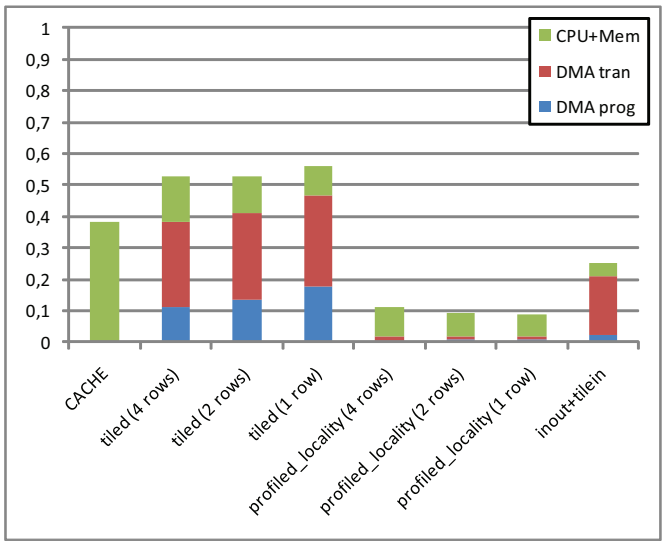


Figure 4.8: Results for LU decomposition benchmark

threads contain a significant amount of work to perform, the cost for DMA is less pronounced, and allows the `tiled` approach to execute 23,60% faster than the cache. This is mainly due to the fact that dynamic scheduling here generate threads that access the main `distributed` array in an irregular fashion. The profile-based approach is insensitive to that, whereas the cache is subject to frequent misses. Employing the `profiled_locality` directive to reduce DMA traffic leads to a 68,92% speedup against the cache. The `inout` approach is not applicable to this benchmark.

4.5.3 Normalized Cut Clustering (NCC)

The main kernel in this application is amenable to dynamic loop parallelization, with independent rows being processed within parallel threads. Pixel data is computed based on a nearest-neighbor pattern on a 5×5 window, thus exhibiting a significant degree of spatial locality of accesses. In this case we expect the cache to be favoured. We consider an array partitioning scheme with the same granularity (i.e. one row per tile), both implicitly with `inout` scheduling, and explicitly with the `tiled` clause. Three arrays are annotated for partitioning and SPM allocation, only 56% of their overall footprint fitting on (the sum of) SPM space, and thus inherently limiting the effectiveness of the `tiled` approach (since part of the array space lays off-chip during the parallel region). This is confirmed by looking at Fig. 4.10, where the `tiled` bars provides the worst

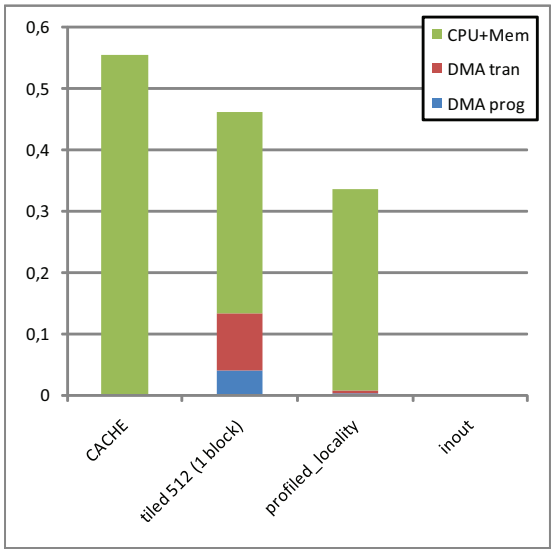


Figure 4.9: Results for Fast Fourier Transform benchmark

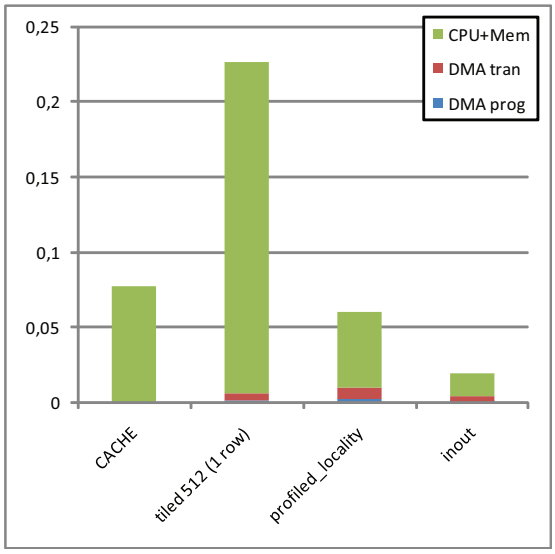


Figure 4.10: Results for Normalized Cut Clustering benchmark

performance (2,93× slower than the cache). Since the arrays are accessed with a regular pattern in time, it is possible to split the parallel region in two smaller regions with the `profiled_locality` directive, so that the entire array subsets accessed by the threads are always on-chip. This solution allows to speed up the execution by 3,77× w.r.t. the `tiled` approach, and by 1,3× w.r.t. the use of the cache. Finally, since the memory access pattern allows the use of `inout` scheduling, iteration-specific fetches of array tiles can be exploited, thus leading to 3,88× speedup w.r.t. the cache.

4.5.4 JPEG Decoding

The JPEG Decoding benchmark features three main tasks, namely Huffman decoding, luminance dequantization and inverse DCT. Huffman decoding is not parallelized, thus is carried out entirely by the master thread. Luminance dequantization can be executed in parallel employing both static or dynamic scheduling. The image is decomposed into 600 DCT blocks (8x8 pixels), each of which can be processed independently by the others. The main parallel loop scans the image blocks, thus we can create threads operating on an arbitrary number of iterations, i.e. working on an arbitrary number of adjacent blocks. When the size of an array tile and the chunk size employed for loop parallelization are equal, we can exploit `inout` scheduling. When employing different

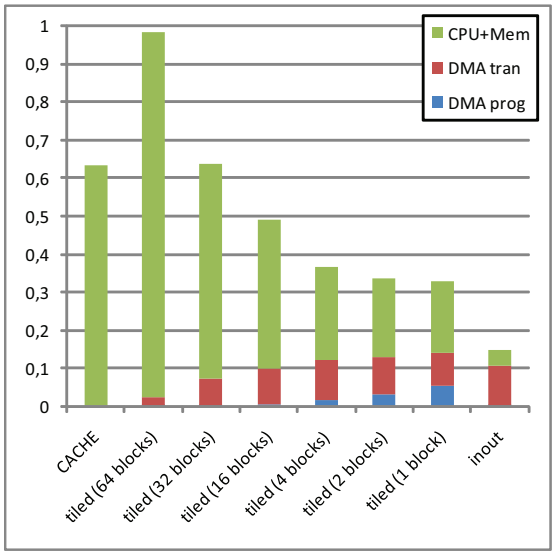


Figure 4.11: Results for the Luminance Dequantization kernel (JPEG benchmark)

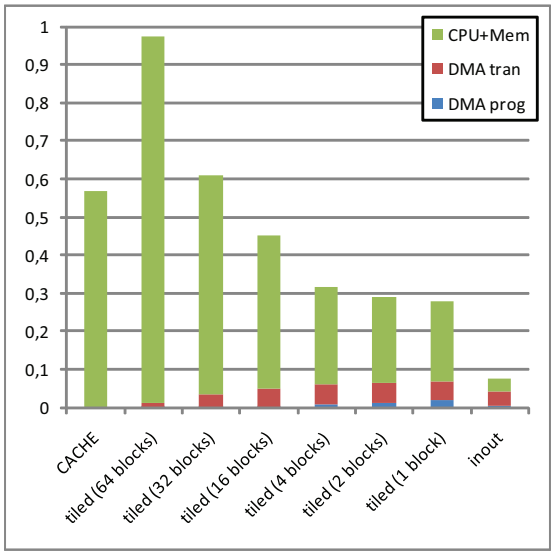


Figure 4.12: Results for the Inverse DCT kernel (JPEG benchmark)

partitioning granularities, the image array can be annotated as `tiled`, and all of the allocation is left to our runtime.

Figure 4.11 shows the results for those two partitioning strategies, where we also provide different granularities for the `tiled` approach. It must be pointed out that the entire image does not fit into the sum of all the SPMs. For this reason, finer partitionig allows a greater number of tiles to be accomodated within SPM space. The simplest partitioning scheme, namely that of dividing the array in as many tiles as available threads, leads to a situation in which a single tile is bigger than a SPM. For this reason our allocator is not capable of placing any subset of the image array on-chip, and always accesses it from external DRAM. This leads to the worst case execution time shown. Reducing the granularity of partitioning allows fitting an incresing number of tiles on-chip. On the other hand, as already noticed with the LU benchmark, initiating a bigger number of transfers to move all tiles to SPM space also increases the cost for DMA programming. In this case this is not a major concern, and employing the finest array partitioning leads to the best performance results for the `tiled` approach, affording a 1,93× speedup w.r.t. the cache. Unsurprisingly, the `inout` approach allows further speedups, since the entire image array is accessed from the SPMs. In this case we achieve a 4,28× speedup.

The results for the IDCT kernel are shown in Fig. 4.12, and basically trace those obtained with the luminance dequantization kernel. This kernel exhibits more computation, as well as a higher number of accesses to neighboring array elements, thus allowing the cache to do slightly better than in the previous kernel. Similarly, the cost for DMA transfers is less predominant with this kernel, thus making the speedup of the `inout` approach against the cache even more relevant (7,63×).

4.6 Conclusion

The emergence of MPSoCs with Explicitly Managed Memories (EMM) raised the necessity for programming models and tools that aid the programmer in the difficult task of achieving performance through efficient exploitation of high-bandwidth, low-latency scratchpad memories (SPM). From one side, efficient exploitation of the hardware resources calls for programming patterns that expose them at the application level. On the other hand higher-level abstractions are desireable for the sake of programming simplicity. In this chapter we described how on-chip SPMs can be easily leveraged from the programming interface by means of simple but powerful extensions to the OpenMP API. Several custom features have been added which allow the programmer to simply express the need for optimized data partitioning and placement through annotations. The details of how partitioning is implemented, however, are hidden from its view and are accomplished by synergistic interaction of profile-based compiler optimization and runtime support for dynamic update of SPM content. In this way at each parallel access to shared arrays every processor references its share of data from its local SPM. Experimental results on several benchmarks confirm the effectiveness of the approach, which allows up to 7,63× speedup against a cache-only solution.

Bibliography

[1] G. Blake, R. Dreslinski, and T. Mudge, "A survey of multicore processors," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 26–37, November 2009. 56

[2] L. Karam, I. AlKamal, A. Gatherer, G. Frantz, D. Anderson, and B. Evans, "Trends in multicore dsp platforms," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 38–49, November 2009. 56

[3] H. woo Park, H. Oh, and S. Ha, "Multiprocessor soc design methods and tools," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 72–79, November 2009. 56

[4] W. Wolf, "Multiprocessor system-on-chip technology," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 50–54, November 2009. 56

[5] Tilera Corp. Product Brief, "Tilepro64 processor," Available: http://www.tilera.com/pdf/ProductBrief_TILEPro64_Web_v2.pdf, 2008. 56

[6] Element CXI Inc. Product Brief, "Eca-64 elemental computing array," Available: http://www.elementcxi.com/downloads/ECA64ProductBrief.doc, 2008. 56

[7] Silicon Hive Databrief, "Hiveflex csp2000 series: Programmable ofdm communication signal processor," Available: http://www.siliconhive.com/Flex/Site/Page.aspx?PageID=8881, 2007. 56

[8] Arm Ltd. White Paper, "The arm cortex-a9 processors," Available: http://www.arm.com/pdfs/ARMCortexA-9Processors.pdf, 2007. 56

[9] NVIDIA Corp. CUDA Documentation, "Nvidia cuda: Compute unified device architecture," Available: http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf, 2008. 56, 59

[10] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementation: a performance view," *IBM J. Res. Dev.*, vol. 51, no. 5, pp. 559–572, 2007. 56

[11] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign.* New York, NY, USA: ACM, 2002, pp. 73–78. 56

[12] P. R. Panda, N. D. Dutt, and A. Nicolau, "On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 3, pp. 682–704, 2000. 56, 61

[13] H. Kim and R. Bond, "Multicore software technologies," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 80–89, November 2009. 56, 58

[14] S. Schneider, J.-S. Yeom, B. Rose, J. C. Linford, A. Sandu, and D. S. Nikolopoulos, "A comparison of programming models for multiprocessors with explicitly managed memory hierarchies," in *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming.* New York, NY, USA: ACM, 2009, pp. 131–140. 56, 58

[15] www.openmp.org, "Openmp c and c++ application program interface v.3.0." 57

[16] A. Marongiu and L. Benini, "Efficient openmp support and extensions for mpsocs with explicitly managed memory hierarchy," in *Design, Automation and Test in Europe Conference and Exhibition, 2009. DATE '09.*, April 2009, pp. 809–814. 58

[17] M. Mehrara, T. Jablin, D. Upton, D. August, K. Hazelwood, and S. Mahlke, "Multicore compilation strategies and challenges," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 55–63, November 2009. 58

[18] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: programming the memory hierarchy," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing.* New York, NY, USA: ACM, 2006, p. 83. 58

[19] S.-Z. Ueng, M. Lathara, S. S. Baghsorkhi, and W.-M. W. Hwu, "Cuda-lite: Reducing gpu programming complexity," pp. 1–15, 2008. 59

[20] R. Chandra, D.-K. Chen, R. Cox, D. E. Maydan, N. Nedeljkovic, and J. M. Anderson, "Data distribution support on distributed shared memory multiprocessors," in *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*. New York, NY, USA: ACM, 1997, pp. 334–345. 59

[21] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, and C. D. Offner, "Extending openmp for numa machines," in *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 48. 59

[22] B. Chapman, F. Bregier, A. Patil, and A. Prabhakar, "Achieving performance under openmp on ccnuma and software distributed shared memory systems," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 8-9, pp. 713–739, 2002. 59

[23] Y. Ojima, M. Sato, H. Harada, and Y. Ishikawa, "Performance of cluster-enabled openmp for the scash software distributed shared memory system," in *CCGRID '03: Proceedings of the 3st International Symposium on Cluster Computing and the Grid*. Washington, DC, USA: IEEE Computer Society, 2003, p. 450. 59

[24] E. Flamand, "Strategic directions towards multicore application specific computing," in *DATE '09*, 2009. 60

[25] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang, "Supporting openmp on cell," in *IWOMP '07: Proceedings of the 3rd international workshop on OpenMP*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 65–76. 60

[26] A. Dominguez, S. Udayakumaran, and R. Barua, "Heap data allocation to scratch-pad memory in embedded systems," *J. Embedded Comput.*, vol. 1, no. 4, pp. 521–540, 2005. 61

[27] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel, "Reducing energy consumption by dynamic copying of instructions onto

onchip memory," in *ISSS '02: Proceedings of the 15th international symposium on System Synthesis*. New York, NY, USA: ACM, 2002, pp. 213–218. 61

[28] S. Udayakumaran, A. Dominguez, and R. Barua, "Dynamic allocation for scratch-pad memory using compile-time decisions," *Trans. on Embedded Computing Sys.*, vol. 5, no. 2, pp. 472–511, 2006. 61

[29] M. Kandemir, J. Ramanujam, M. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "A compiler-based approach for dynamically managing scratch-pad memories in embedded systems," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 23, no. 2, pp. 243–260, Feb. 2004. 61

[30] L. Li, L. Gao, and J. Xue, "Memory coloring: A compiler approach for scratchpad memory management," in *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 329–338. 61

[31] G. Chen, O. Ozturk, M. Kandemir, and M. Karakoy, "Dynamic scratch-pad memory management for irregular array access patterns," in *DATE '06: Proceedings of the conference on Design, automation and test in Europe*. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2006, pp. 931–936. 61

[32] M. Verma, S. Steinke, and P. Marwedel, "Data partitioning for maximal scratchpad usage," in *ASPDAC: Proceedings of the 2003 conference on Asia South Pacific design automation*. New York, NY, USA: ACM, 2003, pp. 77–83. 61

[33] A. Yanamandra, B. Cover, P. Raghavan, M. Irwin, and M. Kandemir, "Evaluating the role of scratchpad memories in chip multiprocessors for sparse matrix computations," April 2008, pp. 1–10. 61

[34] A. M. Abdelkhalek and T. S. Abdelrahman, "Locality management using multiple spms on the multi-level computing architecture," in *ESTMED '06: Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 67–72. 61

[35] M. Kandemir, I. Kadayif, A. Choudhary, J. Ramanujam, and I. Kolcu, "Compiler-directed scratch pad memory optimization for embedded multiprocessors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 12, no. 3, pp. 281–287, March 2004. 61

[36] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor, and J. M. Mendias, "An integrated hardware/software approach for run-time scratchpad management," in *DAC '04: Proceedings of the 41st annual Design Automation Conference*. New York, NY, USA: ACM, 2004, pp. 238–243. 63, 80

[37] S. Martello and P. Toth, "Solution of the zero-one multiple knapsack problem," *European Journal of Operational Research*, vol. 4, no. 4, pp. 276–283, April 1980. [Online]. Available: http://ideas.repec.org/a/eee/ejores/v4y1980i4p276-283.html 73

[38] ——, "Heuristics algorithms for the multiple knapsack problem," *Computing*, vol. 27, pp. 93–112, 1981. 73

[39] M. Loghi, F. Angiolini, D. Bertozzi, L. Benini, and R. Zafalon, "Analyzing on-chip communication in a mpsoc environment," in *DATE '04: Proceedings of the conference on Design, automation and test in Europe*. Washington, DC, USA: IEEE Computer Society, 2004, p. 20752. 79

[40] C. d. S. F. Dorta, A.J. Rodriguez, "The openmp source code repository," *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*, pp. 244– 250, 9-11 Feb. 2005. 80

Chapter 5

Data Mapping for Multicore Platforms with Vertically Stacked Memory

Emerging TSV-based 3D integration technologies have shown great promise to overcome scalability limitations in 2D designs by stacking multiple memory dies on top of a many-core die. Application software developers need programming models and tools to fully exploit the potential of vertically stacked memory. In this chapter, we focus on efficient data mapping for SPMD parallel applications on an explicitly managed 3D-stacked memory hierarchy, which requires placement of data across multiple vertical memory stacks to be carefully optimized. We investigate the applicability of the array partitioning techniques described in the previous chapter to 3D-stacked memory hierarchies. The problem of optimizing array tile placement so as to minimize remote references is even more pressing when dealing with heterogeneous interconnect facilities such as the one we propose here. Vertical (TSV-based) interconnect provides fast access and high bandwidth, whereas accesses to remote memories are transported through the Network on Chip (NOC) in the horizontal plane, thus being subject to increasing access latencies and decreasing bandwidth with the physical distance. This calls for revisitations of the already proposed NUMA-aware placement techniques.

5.1 Introduction

Three-dimensional (3D) stacking technology has recently risen to the research forefront as one of the most high-potential technology innovations for many-core integrated platforms, both in general purpose and embedded computing (11) (12) (6) (4). 3D integration technology provides a number of means to overcome the scalability limitations imposed on many processor designs as 2D technology reaches the nanometer scale. It gives the opportunity to revisit the traditional architectural tradeoffs based on the evidence that the processor and memory sub-systems had to be placed side by side. In 3D stacking they can be placed on top of each other, and linked through vertical interconnects which are more than two orders of magnitude more energy-efficient and denser than the most advanced off-chip I/O channels.

The main benefit of this disruptive technology in high-end embedded computing is to enable the construction of many-core data-processing systems with low latency and high bandwidth access to multiple, large DRAM banks in close spatial proximity. The availability of such an efficient physical layer for processor-to-memory communication and of an enormously increased amount of space in tightly coupled memories will trigger deep changes in high-performance embedded programming. In a nutshell, 3D integration enables distribution in space not only of computation but also of main memory storage to an unprecedented level. Clearly, this brings new distinctive compile- and run-time software development challenges which are just starting to be assessed by the scientific community.

Our first goal is to define a conceptual framework to address these challenges. We model a vertically stacked memory system with the abstraction of *memory neighborhood*: each physical processing element in a large many-core array has fast, large-bandwidth access to a vertical stack of memory banks on top. The processor can also address (in a globally shared memory model) vertical stacks on top of other processors, but corresponding memory transactions will have to be transported through a horizontal on-chip interconnect fabric, typically a Network-on-chip (NoC). This implies a notion of distance: the cost (increased latency and decreased bandwidth) of a memory access sharply increases as we move to memory neighborhoods to far away processors. Fig. 5.1 depicts a high-level view of a 3D integrated architecture and its memory neighborhoods. In this chapter we focus on a concrete embodiment of this model targeting

embedded computing, namely a 3D-integrated platform for multi-dimensional array processing (e.g. antenna arrays, radar images, video images) with explicitly managed data memories.

Typical applications in the domain of array and image processing require the implementation of algorithms for enhancement, analysis, synthesis of multidimensional arrays of "pixel" data. Many of these algorithms are amenable to SPMD (*Single Program, Multiple Data*) parallelization, based on decomposition of data array processing across parallel threads. Mapping this computation model onto a 3D-stacked memory architecture requires careful data placement across multiple physical memories. Placing entire arrays onto a single shared memory encounters scalability problems. Moreover, accessing remote memory stacks induces severe latency overheads. These issues can be efficiently addressed by partitioning data and placing each partition onto the DRAM neighborhood of the processor that mostly references it.

Preliminary results reported in this chapter give clear evidence that 3D-memory aware programming model and application development environment is critically required to achieve high execution efficiency on a vertically integrated embedded multicore platform. The scope of this approach is here precisely limited to SPMD-type parallel applications targeted on a MPSoC with explicitly managed memory hierarchy. Results indicate that a neighborhood-aware software development environment can boost application execution efficiency by up to 6,25×.

5.2 Background and Related Work

Recently, several 3D memory designs have been announced, confirming the benefits of 3D technology for high-efficiency next-generation memory systems (1) (2) (3). The benefits of 3D memories have mostly been explored for high performance systems (4) (5) (3). Kgil et al. (4) present a high performance server architecture where DRAM is stacked on a multicore processor chip. Overall power improvements of $2-3\times$ with respect to a 2D multi core architecture are reported. Similarly, (5) presents a 3D stacked memory architecture for CMP. By changing the internal DRAM architecture (based on true 3D memory organization proposed by (3)), the author claims a 75% speedup. On the industrial front, many companies, including industry leaders IBM and Intel are active in technology and architecture exploration (6) (7) (8) (9).

On the research side, Li et. al investigate in (10) the challenges for L2 design and management in 3D chip multiprocessors. They propose a router architecture and a topology design that makes use of a network architecture embedded into the L2 cache memory. Their term of comparison is 2D NUCA 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 (11). 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 (12), possibly even more than in performance-centric general-purpose computing. In (13), 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 both single-core and multi-core systems show that the 3D placement generates much better results (in terms of data access costs) under the same temperature bounds.

5.3 Target 3D Architecture

The platform template targeted by this work is the 3D-stacked MPSoC depicted in Fig. 5.1. The bottom layer hosts the processing elements of the chip, while the others are composed by DRAM memory banks(5). Each vertical stack features a bank of private memory, only accessible from the local processor, and a bank of shared memory. The collection of all the shared segments is organized as a globally addressable NUMA portion of the address space. In the considered 3D template, memory and CPUs are allocated onto different layers, but our software framework can be applied to different stacking approaches. The bottom layer in Fig.5.1 illustrates the block diagram of the 2D multi core subsystem. It is made by several computational tiles composed by a RISC-like CPU and a small amount of local memory (SPM, caches). Interactions between

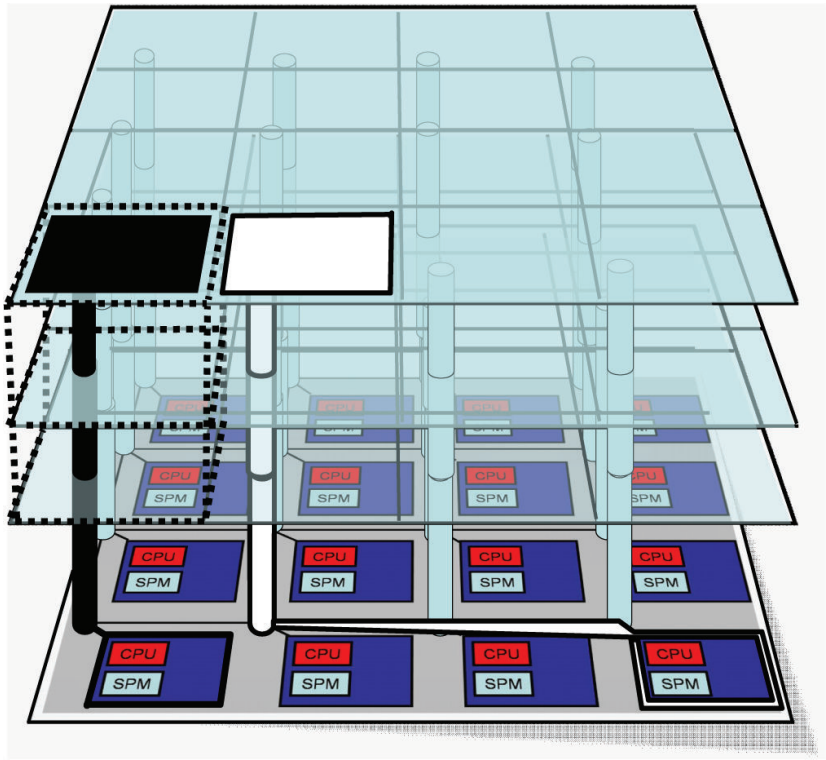


Figure 5.1: Target 3D hardware architecture.

CPUs and memories take place through the overall 3D interconnecting system, which is composed by two main orthogonal and heterogeneous facilities:

- on-layer communication network (NoC), for horizontal communication on the bottom layer
- fast vertical DRAM controller with TSV DRAM physical interface for vertical communication to upper layers.

The whole memory sub-system is accessible from the bottom layer by every tile through this heterogeneous 3D interconnection. Every CPU can reach every SPM memory through the bottom-layer horizontal communication network, but also every sub-bank memory allocated in the upper layers via the on-layer communication network and the appropriate vertical memory controller interface. However, different CPU-to-memory paths have different communication latencies. Shorter paths provide faster communication, while multi-hop paths imply higher latency. Fig.5.1 shows two memory access

examples: the black path denotes a fast communication since the CPU is accessing a memory region which is right above it, while the white path has a higher latency because the transaction has to travel across two links on the bottom layer before reaching the right TSV. Multi-hop transactions are affected by the delay of the links that they need to cross. Moreover, they cause congestion of the overall system interconnect.

5.4 Neighborhood programming

Embedded systems applications can be broadly classified as event-driven (e.g. automotive) or compute and data intensive (e.g. wireless, biomedical, multimedia). Data intensive embedded applications are designed to handle and perform on large data structures. In these programs a high degree of data parallelism is available, where the SPMD execution model is used to replicate the computational kernels in parallel threads working on different subsets of the target data array. In general terms, in the SPMD model each instance of a parallel task (i.e. each thread) accesses only a subset of the shared data. Part of the shared data-set may overlap among different threads, but typically only few border elements are actually accessed by multiple threads. The benefits in placing frequently accessed data close to the processor are well known, particularly when dealing with complex memory hierarchies with NUMA organization. From a practical point of view this has been historically dealt with by means of data transfer to constantly keep in small and fast on-chip SRAM memories such data, with the goal of satisfying most memory references from there.

In 3D architectures, three-dimensional stacking of DRAM (main) memory greatly mitigates the memory size limitation and thus the need for frequent data movements, but it does not solve the problem of efficiently mapping data to physical memory banks. Naive topology-agnostic data placement techniques which allocate entire arrays on a single memory neighborhood may create interconnect bottlenecks and suffer significant latency penalties. Stacking-aware allocation schemes are needed, where different parts of a shared data structure (hereafter called tiles) can be mapped to different physical memory stacks with the goal of minimizing accesses to non-neighbor stacks.

In our view of the neighborhood programming model, the programmer can express at the application level the necessity for distributed placement of a shared data structure. Partitioning is then triggered in the compiler, which transforms the program to

synergistically interact with the runtime environment to find the most convenient placement. We describe in the following how we adapt the OpenMP-based programming framework described in the previous chapter and its support for array partitioning to the 3D MPSoC presented in Sec. 5.3.

5.4.1 Distributed Data Placement

The target 3D MPSoC features both private and shared regions of the address space. Our compiler efficiently deals with thread-private variables by allocating them onto the private block of their host processor's neighborhoods. Dealing with shared data is trickier. The OpenMP memory model assumes a single memory space and provides no facilities to specify how data is to be arranged within the memory space. To specify data placement onto a specific memory neighborhood we extend our custom `distributed` directive.

```
int A[N][M];
#pragma omp distributed (A[, mem_id]);
```

As before, declarations of distributed variables are changed by the compiler into pointers, which point to a region in the shared address space. On the contrary, the optional *mem_id* parameter has here a different meaning. It specifies a target memory neighborhood for placement.

5.4.2 Array Data Partitioning

OpenMP provides work-sharing directives to divide computation among parallel threads, but it lacks means to specify an affinity between the data set touched by a thread and its physical placement on a memory block. We leverage our API extensions to add this feature. The partitioning process can be triggered within a parallel region by annotating the shared array with the custom `split` clause. Similar to our previous 2D implementation, if the `split` clause is specified, the programmer assures that all references issued by a thread fall within a single tile.

```
#pragma omp parallel for split (a)
  for (i=0; i<8; i++)
    a[i] = foo();
```

This functionality is similar to those found in High Performance Fortran (HPF) or OpenMP extensions[18][20] for data distribution. Anyhow, our partitioning technique

is much more flexible, in that – unlike cited approaches – the granularity of array blocking can be arbitrarily small at a very contained cost (see Sec. 5.4.3.2).

Similar parallelization schemes in which each processor touches distinct portions of an array are quite common in OpenMP programs. In this case data tiles are straightforwardly placed close to processors hosting the logically associated thread. In less regular programs often happens that threads need to reference non-local data. Typical implementations of data distribution techniques for NUMA machines rely on heavyweight virtual memory paging techniques to fetch remote data. Here the cost for virtualization is hidden behind the high latencies of the communication medium. On our platform this solution is unfeasible, since we are lacking both the hardware (i.e. MMUs) and software (a full-fledged operating system) support for virtual memory. Furthermore, the high cost for such a virtualization layer would no longer be paid off, due to the low cost for communication in our interconnection system. Thus, our implementation rather relies on a streamlined support for address translation, based on metadata for array indexing that is explicitly managed and allocated (see Sec. 5.4.3). At the application level, we provide the `tiled` clause, that can be coupled with a parallel directive to describe an irregular access pattern. In the following example each array element is placed on a distinct memory, and every thread accesses all of them. Thus, three out of four accesses are to remote memories.

```
#pragma omp parallel tiled (a)
  for (i=0; i<4; i++)
    a[i] = foo();
```

Fig. 5.2 shows how the wanted array element is accessed through address translation. Every single reference to distributed array a is instrumented by our compiler with a call to the library function `GOMP_access_tiled_array()`. Based on the reference offset, the runtime computes a tile ID, then looks up in the `tiles` metadata array for the correct address.

5.4.3 Runtime Support to Data Partitioning and Placement

The partitioning technique described above relies on metadata (the `tiles` array) containing the base address of data tiles. Metadata is replicated onto each processor's memory neighborhood for fast local inspection. By default, arrays are partitioned in

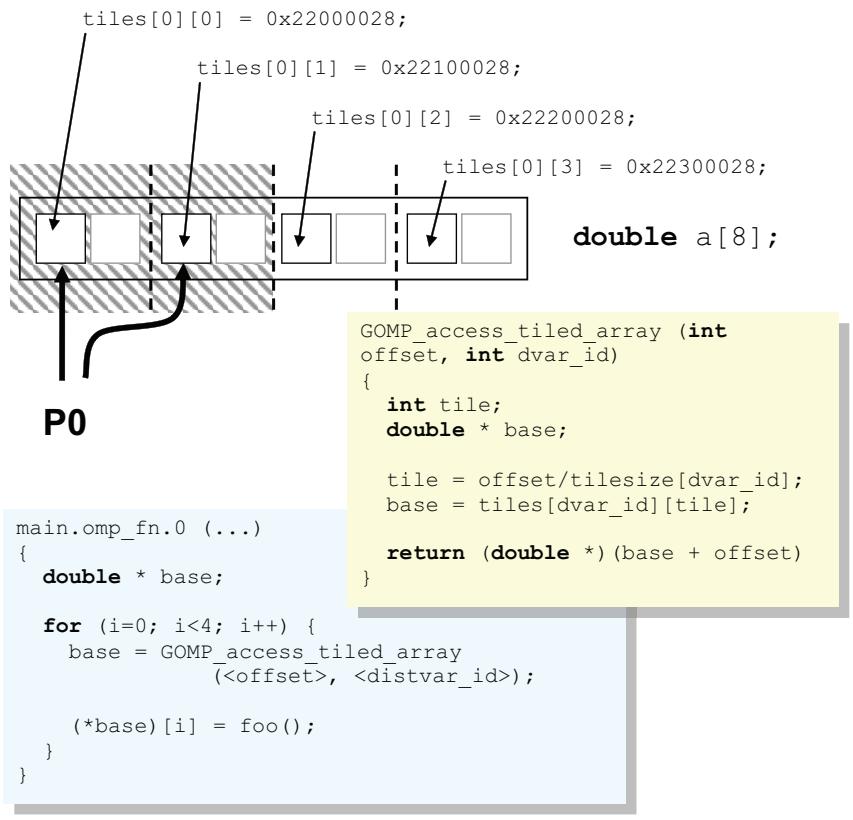


Figure 5.2: Compiler instrumentation of `tiled` arrays

a number of tiles that is equal to the number of worker threads (i.e. cores), and addresses for each tile are generated by the compiler according to a cyclic distribution onto memory neighborhoods. This default choice has three advantages:

1. it captures the thread-to-memory affinity of static loop scheduling (the most common in OpenMP programs).

2. it enables the coarsest partitioning scheme, which generates metadata with very small memory footprint.

3. it requires almost no intervention from the programmer.

The default scheme provides good results for regular applications, that are amenable to static loop parallelization. For programs with strided or irregular memory patterns,

cyclic placement and coarse-grained partitioning may lead to high rates of remote accesses. We show in the following subsections the solutions we provide to improve locality.

5.4.3.1 Automatic Generation of Affinity-based Data Layouts

Default cyclic tile placement can be overridden by providing a custom tile layout descriptor (metadata) in a specific header file. Devising an efficient placement requires insights on the application behavior on memory. To make this task easier we enriched our compilation toolchain with scripts that automatically find affinities between threads and data tiles, based on access count information gathered during a profile run of the application. Metadata representing a placement that minimizes the number of remote references in the program is automatically generated and included for compilation. The algorithm used to accomplish this task is very similar to the one presented in the previous chapter. The main differences reside in a brand new set of architectural parameters:

1. Memory neighborhoods are composed of DRAM memory, as opposed to the SRAM cells employed by scratchpad memories (SPM). Accurately modeling expected access time in the compiler requires considering appropriate latencies for different memory traffic patterns.

2. In our previous implementation of the allocation passes we considered an interconnection medium with uniform latencies (i.e. a crossbar). With this assumption remote SPM accesses are much less sensitive to the physical distance w.r.t. a NoC. In this case the effect of NUMA latencies is much more pronounced, and depends on the actual processor-to-memory communication pattern (see Sec. 5.5.

To conceptually show the benefits of affinity-based placement w.r.t. cyclic placement, let us consider the following example. A loop is statically parallelized among four threads, and default cyclic partitioning is enabled.

```
#pragma omp parallel for schedule(static) split(arr)
  for (i=20; i<N; i++)
    arr[i] = foo();
```

We represent the footprint of threads on the array in Fig. 5.3. Threads are represented with dashed lines, and array tiles with thick black borders. Since the lower boundary

of the iteration space is greater than zero, the portions of the array accessed by the four threads do not overlap completely with the array tiles. In Fig. 5.3.a) we show cyclic (default) tile placement. Color coding is used to associate a tile to a thread, so each tile is associated to a different thread in a cyclic fashion. We use plain and dashed filling to represent local and remote references, respectively. In Fig. 5.3.b) we show affinity-based tile placement. Here tiles are allocated onto the memory neighborhood local to the processor that mostly references it. It is possible to notice that affinity-based placement accounts for the irregularity in loop boundaries. Both tiles 1 and 2 are allocated local to thread 1, and the number of remote accesses is significantly reduced.

5.4.3.2 Refining Partitioning Granularity

Keeping the size of metadata small is profitable when memory space is strictly constrained (e.g. when using SPM space for data allocation). On the other hand, in irregular programs exploiting few large tiles may result in poor approximation of the thread footprint on memory, resulting in poor locality. On our 3D architecture there are no strict memory space constraints, so we can refine the granularity of partitioning. The programmer can specify the number of tiles for partitioning. The finer the partitioning is done, the more overlapping of data tiles with accessed locations is achieved. This is shown in the comparison between coarse-grained affinity-placement (Fig. 5.3.b) and fine-grained affinity-placement (Fig. 5.3.c) for the example loop introduced in the previous section.

In Fig. 5.3.c the array is partitioned in eight tiles, each of which is placed locally to the processor with higher affinity. This significantly reduces the number of remote accesses.

It is worth underlining here that data distribution on traditional NUMA machines either only provide page granularity for partitioning(21) (which is often too coarse to be beneficial), or resorts to very tricky and expensive techniques (e.g. data padding at the page level) to provide finer partitioning(19). We can support fine-grained partitioning with much better efficiency, with arbitrarily small data tiles at the same cost for address translation and at the only increased cost for memory footprint of metadata.

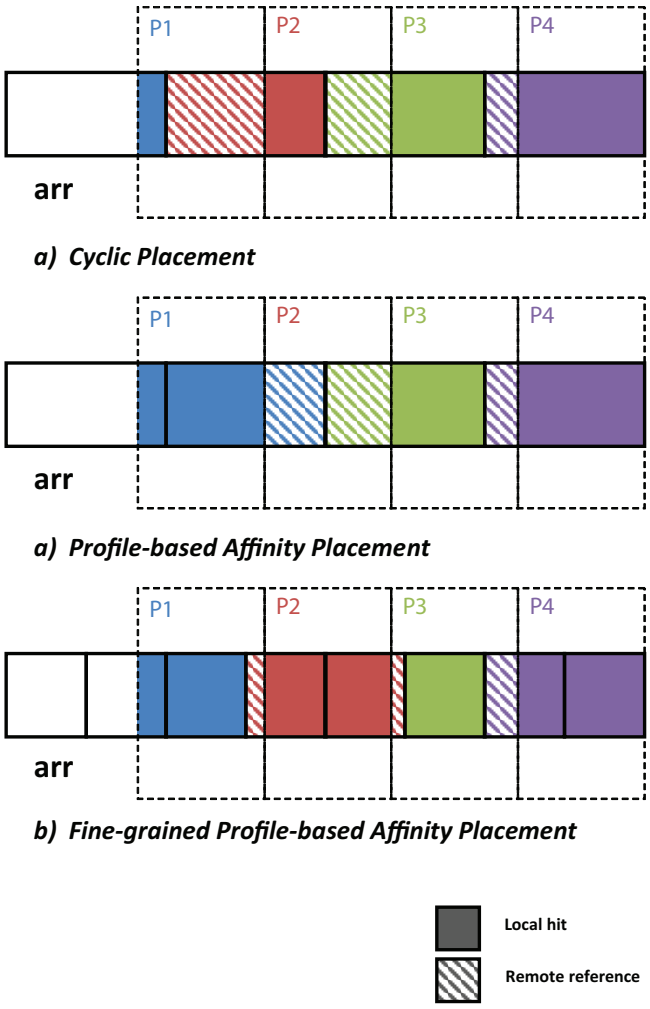


Figure 5.3: Comparison of miss rate for cyclic, coarse-grained affinity-based and fine-grained affinity-based placement

5.5 Experimental Results

We describe in this section the experimental setup used to evaluate our programming framework and the results obtained.

We implemented an instance of the 3D platform template presented in Sec. 5.3 within our SystemC full system simulator. We simulate a 3D chip composed by three layers. The bottom level hosts 16 processor tiles, while memory stacks (16 MB each) reside on the topmost two layers. Each processor tile features a RISC-like CPU coupled with 16KB scratchpad memory (SPM) and a small unified cache (16KB) for private

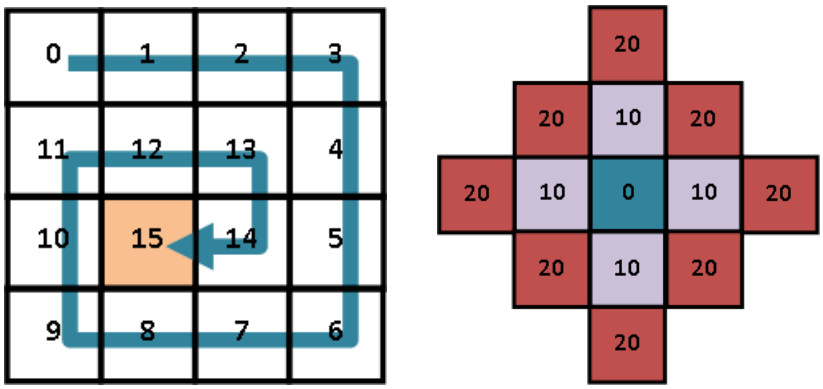


Figure 5.4: Floorplan and scheme for interconnect latency modeling

data and instructions. Caches only manage private data, therefore any coherence issue is prevented. The network on chip on the CMP die is based on the ST STBus protocol.

In the image on the left of Fig. 5.4 we show the layout of processing tiles on the CMP die. Processor IDs increase with the pattern indicated by the arrow. The master core is kept in a central position in the CMP die to balance communication cost among cores towards its memory stack. The memory access time is not constant for the entire hierarchy, but depends on the transaction path. Accesses to local SPM are subjected to only 1 cycle latency. For remote SPMs this cost depends on the internal memory interface latency ($\approx$ 2 cycles), the number of hops to the target memory controller, the contention level on the network, the neighborhood interface latency ($\approx$ 2 cycles), the neighborhood memory latency (1 cycle for SPM, $\approx$ 5 cycles for 3D stacked DRAM).

The zero-load latencies for each core to traverse the interconnect for remote memory access are modeled as depicted in the image on the right. For instance, 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.

We show results obtained with two benchmark applications. The first is a normalized cut clustering (NCC) image processing kernel, and the second is a JPEG decoding algorithm. Performance plots with a breakdown of parallel execution time for each processor (on the X-axis) highlight:

1. **Mem port congestion** - Idle time due to congestion on the memory port (serialization of transactions)

2. **Latency** - Time spent in delivering/retrieving data through the network (zero-load latencies)

3. **CPU time** - Time spent on computation

Such plots are drawn for three program configurations:
Baseline: All shared data is placed in the memory neighborhood of the master processor.
Basic Tiling: Coarse-grained partitioning with affinity-based placement.
Fine-Grained Tiling: Fine-grained partitioning with affinity-based placement.

To further evaluate the effectiveness of the granularity optimization we also provide plots that show the percentage of local memory accesses for decreasing tile sizes.

5.5.1 NCC benchmark

The main program loop is parallelized with static scheduling, but the overall number of iterations does not evenly divide the number of processors. An equal number of iterations is assigned to all processors but the one with the highest ID, which has a lighter workload. This behavior justifies the shorter execution time for processor 15 in the baseline plot (Fig. 5.5).

When all shared data resides in the master core's memory stack – as expected – severe penalties due to memory port congestion are encountered. Latencies to access remote memory neighborhoods also lengthen significantly execution time. The congestion problem is completely removed when applying partitioning, even with coarsest granularity (Fig. 5.6).

This yields a 6× speedup w.r.t. the baseline technique (the plots have different scales). For processors 0, 6 and 12, all memory references are satisfied from the local neighborhood, whereas other processors suffer varying degrees of penalty for accessing remote neighborhoods. Fine-grained partitioning (Fig. 5.6) with eight times smaller tiles allows a further significant reduction of the time spent on the interconnect (an additional 15% speedup). It has to be pointed out that the considered interconnect architecture has very low zero-load latencies, thus limiting the benefits of fine-grained partitioning. We expect it to be even more profitable when considering NoCs with higher zero-load latencies. The cost for fine partitioning is the increased footprint of

metadata in memory. Coarse partitioning employs as many tiles as processors, which requires 64 bytes-metadata. The finest partitioning considered in these experiments generates metadata which has a footprint of 512 bytes (2,5% of the decoded image size). In Fig. 5.11 we plot the percentage of accesses satisfied from local neighborhood.

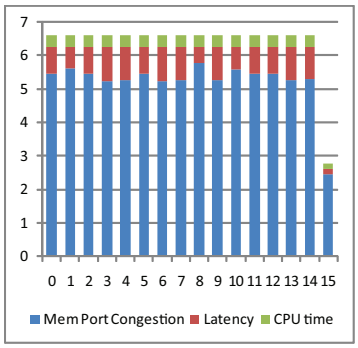


Figure 5.5: Baseline (NCC benchmark). Execution time breakdown.

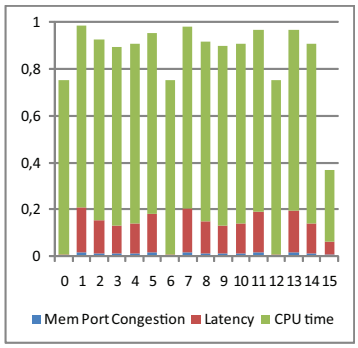


Figure 5.6: Basic tiling (NCC benchmark). Execution time breakdown.

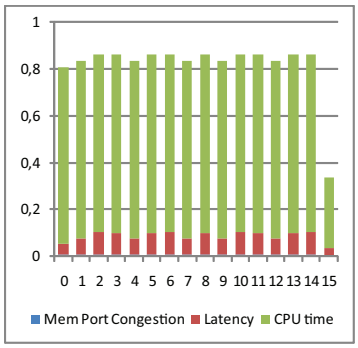


Figure 5.7: Fine tiling (NCC benchmark). Execution time breakdown.

On the X-axis the granularity of partitioning (1 corresponds to the basic technique. 1/2 means tile sizes halved and so on). Locality is improved by 11,43% for array 1 and by 23,60% for array 2 when 1/8 sized tiles are considered.

5.5.2 JPEG decoding benchmark

This benchmark is parallelized with dynamic scheduling. Chunks of iterations are distributed in a FCFS fashion to worker threads. We choose the chunk size to be a fraction (up to 1/8) of the tile size for the coarsest partitioning. This can be considered as a worst-case for the coarse-grained tiling technique. Indeed, even if affinity-based placement allocate tiles close to the processor with highest rate of accesses, still multiple threads insist on the same tile. When the entire image lays un-partitioned in the master processor's memory neighborhood (Fig. 5.9) we experience the usual contention penalties. Due to the high contention on its local neighborhood, the master core is delayed in executing its work. This results in a fewer number of invocations to the runtime library for work assignment, which justifies the shorter time spent on CPU computation.

As expected, coarse-grained tiling (Fig. 5.9) suffers a high number of remote references, which also implies some interconnect congestion. This notwithstanding, a 2,7×

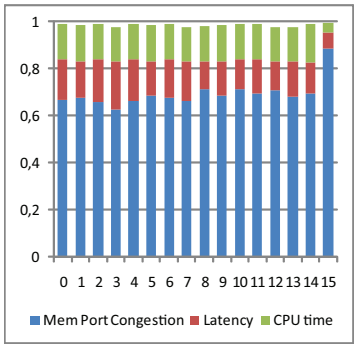


Figure 5.8: Baseline (JPEG benchmark). Execution time breakdown.

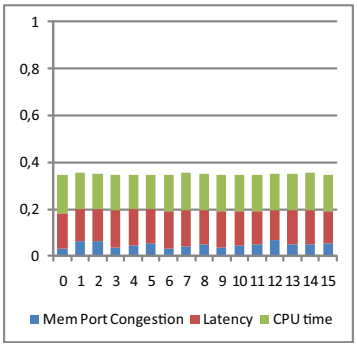


Figure 5.9: Basic tiling (JPEG benchmark). Execution time breakdown.

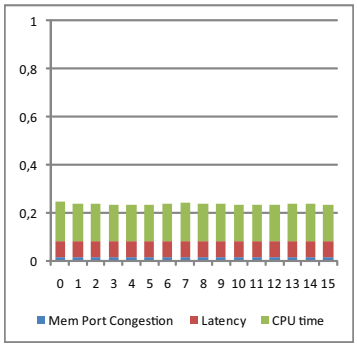


Figure 5.10: Fine tiling (JPEG benchmark). Execution time breakdown.

speedup is achieved w.r.t. the baseline. Refining the granularity at 1/8 tile size leads to 15% reduction of the time spent on memory subsystem (Fig. 5.10), with a metadata footprint on memory which amounts to 1,3% of the image size.

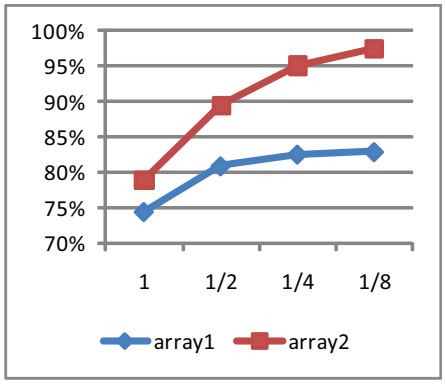


Figure 5.11: % local references with decreasing tile size (NCC benchmark)

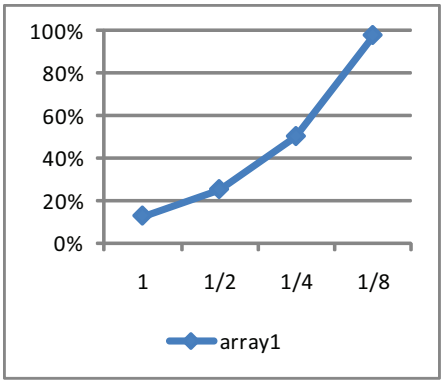


Figure 5.12: % local references with decreasing tile size (JPEG benchmark)

Fig. 5.12 shows that fine-grained partitioning allows almost perfect overlapping of thread and data space, thus leading to excellent locality.

5.6 Conclusion

In this chapter we moved a first step toward the definition of 3D-aware programming abstractions and tools to enable effective exploitation of the large potential for increased computational efficiency offered by 3D-integrated memory architectures. We outlined

the concept of memory 3D neighborhood programming, and we developed language extensions, compiler enhancements and run-time support for neighborhood programming within the standard OpenMP shared memory programming environment.

The approach is specifically focused to explicitly-managed memory architectures and applications with SPMD parallelism. Results demonstrate that array partitioning techniques are extremely important to achieve performance on such a NUMA machine, and that our lightweight compiler support for metadata-based address translation allows interesting speedups even for fine grained parallelization schemes.

Much work remains to be done both in advanced optimization techniques and in extending the scope of applicability of the memory neighborhood concept to other classes of architectures (e.g. cache-coherent SMP) and application classes with different forms of parallelism.

Moreover, while for 2D architectures with SPM we were able to develop language and compiler support for dynamic data movement through DMA, it is possible that with 3D MPSoCs this will no longer be an efficient solution. Indeed, updating the content of small SPMs does not bring much transfer overhead, but with 3D-stacked DRAM the amount of tightly coupled memory for data management is increased to an unprecedent level. Dynamically swapping the content of memory banks to capture the access pattern in the program may require much too big transfers, thus leading to reduced performance and increased energy consumption. In this context it may be more profitable to statically determine a starting data partitioning scheme and then dynamically adapt the workload so as to assign threads to processors with maximum affinity. Data should never be moved, whereas threads should be allowed to migrate when the access pattern in the program changes. Ideas from work-stealing and dynamic loop parallelization techniques could be borrowed to investigate this approach.

5. DATA MAPPING FOR MULTICORE PLATFORMS WITH VERTICALLY STACKED MEMORY

Bibliography

[1] U. Kang and et al. 8gb 3d ddr3 dram using through-silicon-via technology. In *ISSCC '09* 2009. 97

[2] M. Kawano et al. Three-dimensional packaging technology for stacked dram with 3-gb/s data transfer. In *IEEE TED*, 2008. 97

[3] 3d stacked ddr2 sdram *www.tezzaron.com/memory/TSC_Leo_II.htm*. 97

[4] T. Kgil et al. Picoserver: Using 3d stacking technology to build energy efficient servers. In *J. Emerg. Technol. Comput. Syst.*, 2008. 96, 97

[5] G. H. Loh. 3d-stacked memory architectures for multi-core processors. In *SIGARCH Comput. Archit. News*, 2008. 97, 98

[6] P. G. Emma and E. Kursun. Is 3d chip technology the next growth engine for performance improvement? In *IBM Journal of Research and Development*, 2008. 96, 97

[7] B. Black et al. Die stacking (3d) microarchitecture. In *MICRO 39*. 97

[8] Li and et al. 3d integration: Opportunities and challenges. In *ISCAS '08*, 2008. 97

[9] Made in ibm labs: Ibm cools 3-d chips with h2o *www-03.ibm.com/press/us/en/pressrelease/24385.wss*. 97

[10] F. Li et al. Design and management of 3d chip multiprocessors using network-in-memory. In *ISCA '06*, 2006. 98

[11] S. Gu et al. Stackable memory of 3d chip integration for mobile applications. In *IEDM '08*, 2008. 96, 98

[12] M. Facchini et al. System-level power/performance evaluation of 3d stacked drams for mobile applications. In *DATE '09*, 2009. 96, 98

[13] O. Ozturk et al. Optimal topology exploration for application-specific 3d architectures. In *ASP-DAC '06*, 2006. 98

[14] J.H. Kelm et al. SChISM: scalable cache incoherent shared memory CRHC-08-06.

[15] K. Fatahalian et al. Sequoia: programming the memory hierarchy. In *Supercomputing 2006*, 2006.

[16] T. J. Knight et al. Compilation for explicitly managed memory hierarchies. In *PPoPP '07*, 2007.

[17] W.W. Carlson et al. Introduction to UPC and language specification CCS-TR-99-157.

[18] R. Chandra et al. Data distribution support on distributed shared memory multiprocessors In *PLDI '97*, 1997.

[19] J. Bircsak et al. Extending OpenMP for NUMA machines. In *Supercomputing 2000*, 2000. 105

[20] B. Chapman et al. Achieving performance under OpenMP on ccNUMA and software distributed shared memory systems. In *Concurrency and Computation: Practice and Experience 14*, 2002.

[21] Y. Ojima et al. Performance of cluster-enabled OpenMP for the SCASH software distributed shared memory system. In *CCGRID '03*, 2003. 105

[22] E. Flamand. Strategic directions towards multicore application specific computing. In *DATE '09*, 2009.

Chapter 6

OpenMP Support for NBTI-induced Aging Tolerance in MPSoCs

Aging effect in next-generation technologies will play a major role in determining system reliability. In particular, wear-out impact due to Negative Bias Temperature Instability (NBTI) will cause an increase in circuit delays of up to 10% in three years(8). In these systems, NBTI-induced aging can be slowed-down by inserting periods of recovery where the core is functionally idle and gate input is forced to a specific state. This effect can be exploited to impose a given common target lifetime for all the cores.

In this chapter we present a technique that exploits and extends the OpenMP parallel programming model to allow core-wear-out dependent insertion of recovery periods during loop executions so that the wear-out process can be finely controlled. At this level, performance loss can be compensated based on the knowledge of recovery periods. Loop iterations are re-distributed so that cores with longer recovery will be allocated less iterations.

6.1 Introduction

Embedded multiprocessor systems-on-chips (MPSoCs) fabricated in upcoming nanometer technologies will be increasingly affected by aging mechanisms leading to threshold voltage increase (9) which implies circuit slowdown. As a consequence, guardbands

(GB) are inserted to compensate for circuit delay. These guardbands will shrink during core activity until their complete consumption will lead to timing violations. In absence of correction mechanisms, these violations will determine system failure. With respect to single core systems, in multicore platforms an additional reliability issue is that both the initial GB margin and its consumption rate are not uniform across the cores. As a consequence, to prevent the less reliable core to dictate the overall system lifetime, the GB consumption must be equalized as much as possible. At system level, this can be obtained by monitoring the guardband consumption (2) (4) and slowing down the aging process of less reliable cores (16).

The strategy to slowdown aging of cores depends on the considered aging effect. The main aging phenomena affecting nanometer devices are Negative Bias Temperature Instability (NBTI) and Hot Carrier Injection (HCI), for which wear-out takes place only during activity periods. In particular, NBTI has gained much attention from recent research because it is considered a dominant effect (10). NBTI is due to the dissociation of Si-H bonds along the silicon-oxide interface in presence of a negative bias ($V_{gs} = -V_{dd}$) on PMOS transistors, which causes the generation of traps. These traps lead to the increase in the threshold voltage. Recent studies demonstrate that NBTI will be relevant in forthcoming technologies, leading to up to 10% voltage increase in three year lifetime (8).

The NBTI degradation model is characterized by a recovery effect, caused by the reduction of interface traps when the negative bias is removed. As a result, the threshold voltage decreases. Thus, NBTI-induced aging can be partially compensated by imposing a virtual ground (i.e. a logical "1") to PMOS transistors gates for a certain period of time, namely the recovery period, where the core is idle from a functional viewpoint. As a result, it is possible to slow-down GB degradation by interleaving normal core activity with idle periods where the core can be placed in a recovery state. The impact of NBTI does not depend on the granularity and distribution of stress/recovery periods but only on their total duration (11). This allows to efficiently distribute the required idleness with convenient granularity. This can be flexibly tuned to match the characteristics of the workload/programming model chosen to parallelize the target application. The programming model ultimately reflects the features of the underlying hardware platform.

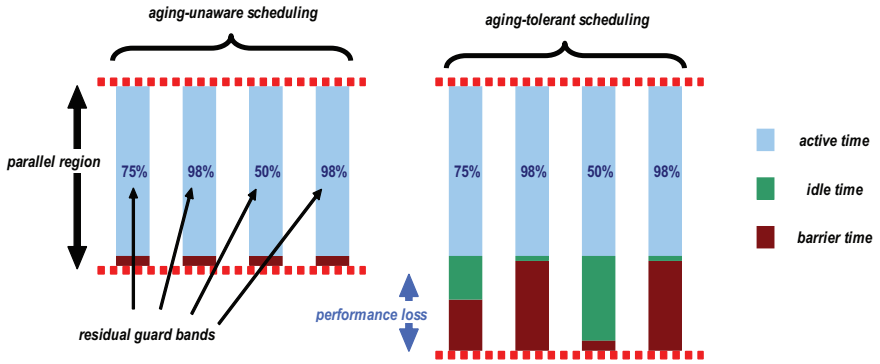


Figure 6.1: Performance loss to support aging-tolerant loop parallelization.

In this chapter we consider embedded MPSoCs for data-intensive processing under the SIMD (Single Instruction Multiple Data) execution model. Aging issues in this kind of platforms can be very critical since they are intensively used during their lifetime, so techniques to hide the effects of aging are desirable. Applications running on these systems focus on a very common data parallel scenario where each core works on a portion of a data structure (e.g. array or matrix) and must synchronize with the others on a barrier. Similar parallelization schemes are typically focused on parallel loops, whose iterations are spread among several concurrent threads. OpenMP is the de-facto standard for such a parallel execution model. In the OpenMP model idleness insertion can be managed at the granularity of a single iteration (or chunks of iterations). This choice allows very fine control on the actual duration of idle and active periods, and thus on the entity of stress and recovery phenomena applied to cores.

Idleness insertion impacts workload balancing because of non-uniform GB consumption rates. Starting from a balanced workload distribution, the addition of idleness increases the overall execution time. In barrier-based parallelization schemes, the overall lengthening of the parallel region – hereafter indicated as *performance loss* – is dictated by the more degraded core (i.e. the one with the longest idle period). This situation is depicted in Figure 6.1. Residual guard bands are indicated as percentages. Longer idle periods are allocated to processors with smaller GB.

The impact of idleness on loop execution time can be evaluated so that iteration redistribution among the cores can be exploited to minimize it. More precisely, perfor-

mance loss can be compensated by proper re-allocation of workload to cores depending on the idleness distribution. The compiler can allocate less iterations to cores with smaller guard bands (and longer idle periods).

The proposed workload re-allocation strategy has been integrated within the GCC OpenMP(GOMP) compiler. The OpenMP interface has been extended with custom clauses to be coupled with the worksharing directives. These clauses augment the existing static and dynamic parallelization schemes with aging-tolerant scheduling facilities. The execution time of each iteration is retrieved through compiler-inserted profiling instructions. Based on this information, actual idleness distribution is achieved by interacting with the runtime library. The library API has been extended with functions that support duty cycling and partitioning algorithms aimed at minimizing performance loss.

6.2 Background and Related Work

Aging problems can be tackled at various abstraction levels, ranging from transistor level, architectural and system software level. Software approaches are very attractive because they can exploit workload knowledge to reduce the performance impact of these techniques. A common purpose of various approaches recently proposed is to provide wanted performance and match real-time constraints through statistical scheduling (17) or learning algorithms (18). In (15) Roberts et al. present a scheduling approach which is aimed at recovering the performance impact due to non-uniform chip degradation. They propose an integer linear programming method to determine an optimal scheduling for streaming applications. Moreover, task migration is also considered as solution to handle the time dependent effect of wear-out. A complete framework, called *Facelift*, performing scheduling and voltage scaling to slow down aging is presented in (16). It exploits a non linear optimization strategy to find the optimal scheduling and voltage changes.

Comparing to this work, our techniques are focused on compiler-level strategies, and for this reason we implement aging tolerance at the parallel application level and not at the operating system level. As explained in Chapter 3, most MPSoCs feature heterogeneous runtime support on different processing tiles. Consequently, typical

MPSoC-suitable OpenMP implementations are OS-less. Moreover, our approach reduces the performance hit by playing with loop iteration re-scheduling, which is not possible at the OS-level, where tasks are given. A similar approach is taken by authors of (5). They propose a variability-aware algorithm that maps computations onto available processors so that each processor runs at its peak frequency rather than simply using chip-wide lowest frequency amongst all cores and highest cache latency. Unlike ours, this technique aims at maximizing performance, but does not cope with wear-out phenomena in any manner. In presence of aging, exercising processors with different degrees of GB consumption at the same rate (i.e. at their peak performance) leads to a situation in which the most degraded core dictates overall system lifetime.

6.3 Aging Model and Idleness Constraints

Multicore designs in current technologies suffer significant within-die process variation, thus leading to nominally identical processors supporting non-homogeneous maximum frequencies. Furthermore, during processor service life stresses induced on transistors by normal switching activity results in gradually slower critical paths. In order to meet system lifetime constraints, designers add timing guardbands to their designs to absorb any increase in critical path delay. One conservative approach to deal with this source of heterogeneity, which is often employed to simplify the design, is to use a single frequency domain where the slowest core determines the frequency of the whole chip. Moreover, if processors are exercised at a similar rate, the slowest core will consume its own guardband earlier than the others. These effects can strongly impact system lifetime and for this reason an increasing effort is put at the various layers of MPSoC design to detect and compensate them. Designers implemented delay monitors (4) (2) spread across the chip that provide degradation information in terms of circuit delay, from which the guardband consumption can be derived. As such, the guardband size provides a upper bound on the allowed ΔV_{th} for each core.

Based on this information, our objective is to equalize GB consumption time among the cores. In principle, we can set a predefined target lifetime, which would be equal for all the cores. In order to achieve a wanted target lifetime, we need to slow down aging rate for less reliable cores (the ones with smaller GB). For NBTI-induced aging it is possible to slow-down core degradation by imposing idle periods. In these periods,

if the core is set into a particular state (recovery state) where the gates of PMOS transistors are tied to a virtual ground (i.e. a logical "1" is applied) the threshold voltage degradation is partially recovered. The increase in V_{th} during the stress phase can be modeled as follows (16):

$$\Delta V_{th,stress} = A_{NBTI} \cdot t_{ox} \cdot \sqrt{C_{ox}(V_{dd} - V_{th})} \cdot \exp^{(\frac{V_{dd}-V_{th}}{t_{ox}E_0} - \frac{E_a}{kT})} \cdot t_{stress}^{0.25} \tag{6.1}$$

where t_{stress} is the time under stress, t_{ox} is the oxide thickness and C_{ox} is the gate capacitance per unit area. E_0, E_a and k are constant equal to $0.2V/nm$, $0.13eV$ and $8.6174 \cdot 10^{-5}eV/K$ while A_{NBTI} is a constant dependent on the aging rate. The recovery phase is governed by the following equation:

$$\Delta V_{th} = \Delta V_{th,stress} \cdot (1 - \sqrt{\eta \cdot \frac{t_{rec}}{t_{stress} + t_{rec}}}) \tag{6.2}$$

where t_{rec} is the time under recovery and η is a constant equal to 0.35. Depending on the guardband value we can compute the maximum ΔV_{th}^i each core i can accommodate before failing. The relationship between ΔV_{th}^i and the guardband value GB^i is given by the following standard switching delay expression:

$$T_s^i = \frac{V_{dd} L_{eff}}{\mu (V_{dd} - V_{th}^i)^\alpha} \tag{6.3}$$

Now, since $T_s^i = DCP^i + GB^i$ where DCP^i is the initial delay critical path of core i, we can compute the guardband size as a function of the threshold voltage:

$$\Delta V_{th}^i = V_{th}^{init,i} - V_{th}^{stress,i} \tag{6.4}$$

where $V_{th}^{init,i}$ is the voltage threshold corresponding to the initial critical path delay DCP^i, while $V_{th}^{stress,i}$ is the maximum voltage threshold corresponding to the largest allowed delay (i.e. guardband fully consumed). Thus we can substitute delay expressions into this equation to obtain the maximum allowed voltage increase for each core as a function of its current GB:

$$\Delta V_{th}^{max,i} = f(GB^i) \tag{6.5}$$

On the other side, Eq. (6.1) allows to express the voltage increase as a function of the stress and recovery time:

$$\Delta V_{th}^{i} = f(t_{stress}^{i}, t_{rec}^{i}) \tag{6.6}$$

Combining (6.1) and (6.4) and considering a given target lifetime:

$$t_{life} = t_{stress}^{i} + t_{rec}^{i} \tag{6.7}$$

we can compute the amount of recovery time t_{rec}^{i} needed to consume ΔV_{th}^{i} in a time t_{life}, the same for all the cores. The recovery time obtained in this way can be used to compute the percentage of idleness I^i to be allocated to maintain the wanted target lifetime:

$$I^i = t_{rec}^{i}/t_{life}. \tag{6.8}$$

Cores having larger GBs, whose values can be read from circuit monitors, will be allocated less idleness. Monitors can be implemented either using hardware circuits to measure circuit delays (2) or by monitoring activity (stress) periods and using an analytical model to compute the related circuit delay increase.

The OpenMP extensions we developed leverage this information to perform idleness distribution and iteration allocation at each loop execution to the cores depending on their GB values. We refer to the percentages of idleness needed on different cores to compensate for aging effects as “aging indexes”. Aging indexes are computed based on the formulas described above, and can be inspected by the runtime environment to take decisions on workload and idleness distribution. More precisely, we read aging indexes at each loop execution. Based on this feedback, we tune the amount of work on each core by means of a custom partitioning algorithm (see Section 6.4), and allocate a corresponding recovery period, so that the wanted lifetime is respected.

6.4 Aging-aware OpenMP Support Implementation

The basic directive provided by the OpenMP API for specifying parallel execution within the code is `#pragma omp parallel`. As explained in Chapter 3, enclosing a portion of code within the scope of this directive allows the programmer to identify a parallel task, and instructs the compiler to generate code to fork worker threads onto which the parallel task is mapped. The use of this directive is typically coupled with one of the two work-sharing directives, `#pragma omp for` and `#pragma omp sections`. The former enables data parallelism by partitioning the iteration space of a for loop between worker threads, whereas the latter leverages task parallelism. The OpenMP work-sharing model provides means to achieve balanced execution among processors by outlining parallel tasks containing similar amounts of work.

The basic idea of our aging-tolerant policies is that of lengthening the lifetime of degraded cores to match that of the most reliable core, thus meeting expected system service life. This is achieved through explicit insertion of idleness periods, which are interleaved with normal activity. The granularity at which we perform duty cycling (i.e. the duration of active periods) is specified by the use of a particular work-sharing directive. For task parallelism the granularity is that of the task itself, whereas for data parallelism the granularity may be that of a single iteration, or of a chunk of iterations. The compiler inserts time sampling instructions at the beginning and at the end of the work block (with the discussed granularity), then instantiates a call to the custom `omp_sleep` library function passing it the profiled execution time of the work block. The sleep time is a function of the execution time and the aging of the target processor. Information on the aging of each processor is embedded within specific metadata in the custom OpenMP runtime environment. This "aging index" is as a number between 0 and 1, which expresses the percentage of idleness needed on a core to compensate for its degradation. It can be inspected whenever needed through a call to the custom `omp_get_aging_index` function, which implements the aging model described in Section 6.3.

The described mechanism efficiently augments OpenMP with an infrastructure for duty cycling. Processors with different aging indexes require different sleep times, thus leading to parallel tasks with non-homogeneous duration and finally implying unbalanced execution. While the balancing issue can be easily addressed by integrating

our duty cycling mechanism with the runtime support for dynamic scheduling (see Sec. 6.4.2), things get more complicated when dealing with static scheduling (see Sec. 6.4.1). The `schedule(static)` clause is useful when parallelizing loops whose iterations have roughly the same duration, since it affords good balancing with very small scheduling overhead w.r.t. dynamic approaches. Furthermore, smart combination of static scheduling and chunking is the only means provided by the OpenMP API to achieve good data locality. For this reason it is very important to consider static scheduling in our aging-tolerant framework. As described in Section 6.1 and shown in Figure 6.1, simply inserting idle periods in presence of static scheduling would lead to very unbalanced overall loop execution time. The barrier implied at the end of the parallel region forces all cores to wait for the less reliable, thus leading to the highest performance degradation.

In the following sections we present custom extensions to the OpenMP API that allow to efficiently address this issue and reduce performance loss.

6.4.1 Static Scheduling

The simplest algorithm to parallelize a *doall* loop is that of evenly dividing its iteration space among available worker threads. OpenMP allows to do it with the use of the `schedule(static)` clause combined with the `for` directive:

```
#pragma omp parallel for schedule(static)
  for (i=0; i<N; i++)
    { /* body */ }
```

The compiler transforms the loop so that lower and upper bounds are computed locally by each thread, based on the number of concurrent workers and on their IDs.

```
int nthreads = omp_get_num_threads();
int tid = omp_get_thread_num();
int chunk = N/nthreads;
int LB = tid * chunk;
int UB = (tid + 1) * chunk;

for (i=LB; i<UB; i++)
  { /* body */ }
```

As discussed in the previous section, duty cycling helps in achieving homogeneous guard-band consumption, but introduces imbalance. To achieve load balancing while

hiding the effects of aging on system lifetime, we replace the original partitioning algorithm in the compiler with a simple yet effective aging-aware scheduling technique. In what follows the number of iterations (W_i) needed to equalize execution time (T_i) of all the cores is computed.

Let us consider the following parameters:

N	Total loop iterations
M	Number of processors
A_i	Aging index for the i-th core
ΔT	Iteration duration
W_i	Work iterations for i-th core

Overall work time for the i-th core can be expressed like

$$T_{W,i} = \Delta T \cdot W_i \tag{6.9}$$

Total loop time for processor i is expressed by the formula

$$T_{T,i} = T_{W,i} + T_{S,i} \tag{6.10}$$

where the sleep time is a function of the active time and the aging index

$$T_{S,i} = (1 - A_i) \cdot T_{W,i}$$

which can be substituted in (6.10)

$$T_{T,i} = (2 - A_i) \cdot T_{W,i}$$

Loop execution time must be balanced between cores, namely

$$T_{T,i} = T_{T,j} \qquad \forall i, j$$

At system startup, our runtime inspects the reliability of each core, and designates the most reliable processor as the master (hereafter core M). Sleep times are normalized to that of the less degraded core M, so we consider

$$T_{T,M} = T_{W,M}$$

and express the number of iterations of each slave core as a fraction of the iterations of the master (M) core

$$T_{W,M} = T_{T,i} = (2 - A_i) \cdot T_{W,i} \qquad \forall i \in [1, M)$$

$$W_M \cdot \Delta T = (2 - A_i) \cdot W_i \cdot \Delta T$$

$$W_i = \frac{W_M}{(2 - A_i)} \tag{6.11}$$

The iterations of the master core can be computed by balancing the iterations

$$\sum_i W_i = N \;\Rightarrow\; W_M + \sum_{i=1}^{M-1} \frac{W_M}{(2 - A_i)} = N$$

which finally leads to

$$W_M = \frac{N}{1 + \sum_{i=1}^{M-1} \frac{1}{(2-A_i)}} \tag{6.12}$$

Having W_M we can compute W_i using eq.6.11.

The aging-aware partitioning algorithm is triggered by the use of the custom `schedule(static_rel)` clause

```
#pragma omp parallel for schedule(static_rel)
  for (i=0; i<N; i++)
    { /* body */ }
```

The compiler has been customized to emit the following parallel code

```
int tid = omp_get_thread_num();
omp_part_iteration_space(N, tid);
int LB = omp_get_lower_bound(tid);
int UB = omp_get_upper_bound(tid);
long start, stop;
  for (i=LB; i<UB; i++)
    {
        start = omp_get_wtick();
        /* body */
        stop = omp_get_wtick();
        omp_sleep(stop-start);
    }
```

Lower and upper bounds for each thread are no longer computed locally, but rather retrieved through calls to the runtime library. Timestamp sampling instructions are inserted to compute the duration of the loop body, which is then passed to the runtime to force the needed amount of idleness.

The `omp_part_iteration_space()` function implements our aging-aware partitioning algorithm. Every thread inspects its aging index, then the master core executes the partitioning algorithm. Slave cores wait on a barrier for lower and upper bounds to be computed for every thread. After this synchronization step, every thread retrieves its chunk of the original iteration space through a call to the custom `omp_get_lower_bound()` and `omp_get_upper_bound()` functions.

The extensions to the OpenMP library (libgomp) are summarized in Tab. 6.1.

6.4.2 Dynamic Scheduling

Non-uniform duration of different loop iterations can lead to load imbalance issues when using static scheduling schemes. To deal with this problem OpenMP provides a `schedule(dynamic, chunk)` clause. The programmer decides the granularity at which the scheduler is invoked by specifying a *chunk* size. This parameter represents the number of loop iterations that are folded within a single task. Each thread participating in a dynamically scheduled parallel loop continuously invokes the runtime to obtain the next available work chunk.

When enhancing dynamic scheduling scheme to support duty cycling we no longer need to cope with load balancing issues, since lengthening the execution time of a thread by inserting idle periods has a side effect of having it invoke the scheduler less frequently. More reliable cores will instead increase the number of requests for chunk assignment, thus "stealing" part of the iterations originally assigned to degraded processor. To adapt the framework to support duty cycling one possible solution is that of profiling execution time at chunk granularity.

The use of the custom `schedule(dynamic_rel[, chunk])` clause

```
#pragma omp parallel for schedule(dynamic_rel,20)
  for (i=0; i<N; i++)
    { /* body */ }
```

instructs the compiler to generate code that calls custom versions of the library functions for dynamic loop scheduling, namely `GOMP_loop_dynamic_start_rel()` and `GOMP_loop_dynamic_next_rel()`. The scheduling algorithms in these custom functions do not introduce any changes with respect to the original, but they are enhanced with execution time profiling instructions. The collection of timestamps for execution time

profiling at the granularity of a single chunk is performed within these functions. Idleness insertion is forced at every scheduling event, namely every time that a thread queries the runtime for another chunk of iterations to process. Once the duration of the chunk has been retrieved, it is passed to the `omp_sleep` function actual insertion of idleness.

`void omp_part_iteration_space (int iterations, int pid)`	Computes lower and upper bound for each thread participating in a parallel loop. Boundaries are computed exploiting our partitioning algorithm and stored in library metadata.
`int omp_get_lower_bound (int pid)`	Returns lower bound for thread `pid`'s iteration space.
`int omp_get_upper_bound (int pid)`	Returns upper bound for thread `pid`'s iteration space.
`GOMP_loop_dynamic_start_rel(...)`	Initializes metadata for aging-aware dynamic scheduling.
`GOMP_loop_dynamic_next_rel(...)`	Dynamically schedules next work chunk in an aging-driven manner.
`int omp_get_wtick (void)`	Returns current timestamp.
`int omp_get_aging_index (int pid)`	Returns current aging index for processor `pid`.
`int omp_sleep (long cycles)`	Forces wanted idleness on the caller core based on its aging index and on the profiled execution time.

Table 6.1: API extensions to support aging-tolerant scheduling

6.5 Experimental Setup and Results

The MPSoC architectural template that we target in this work is composed by 16 RISC-like processing elements, each featuring private L1 instruction and data caches as well as a scratchpad memory. On-chip shared and private memories are accessed through a shared bus (amba AHB), and synchronization facilities are provided by a hardware semaphore device. We assume all cores to operate at the same frequency. Aging indexes are implemented as special registers that are periodically updated by the aging model, and that can be inspected by the software library.

Table 6.2 lists the benchmarks used to conduct the experiments. We provide four classes of results and plots to highlight:

- **Overhead**: A breakdown of the sources of overhead in our algorithms
- **Idleness**: The precision of our technique in distributing the wanted amount of idleness

Benchmark	Source
LU Decomposition (c_lu)	OmpSCR(3)
Loops with dependences (c_loop_dep)	OmpSCR
Jacobi (c_jacobi)	OmpSCR
Computing Π (c_pi)	OmpSCR
Mandelbrot set area (c_mandel)	OmpSCR
Embarassing parallel (ep)	NAS Parallel Benchmarks(6)

Table 6.2: Benchmarks

- **Balancing**: The load balancing achieved by our partitioning schemes w.r.t. the original application
- **Performance**: The effectiveness of our scheduling policies in minimizing the performance loss due to duty cycling

For each benchmark we provide results for the following program configurations:

- ***static***: The program is parallelized with the original OpenMP `static` clause. There is no awareness of platform aging at the software level.
- ***static + sleep***: The program is parallelized with the original OpenMP `static` clause, but the framework is aware of system aging, and is augmented with duty cycling. No aging-aware workload distribution policy is enabled, thus leading to worst-case performance loss.
- ***static_rel***: The program is parallelized with the custom `static_rel` clause. Aging-aware loop partitioning takes place.
- ***dynamic***: The program is parallelized with the custom `dynamic_rel` clause to deal with non-uniform duration of loop iterations. Dynamic scheduling of iterations is augmented with duty cycling.
- ***chunked***: The program is parallelized with the custom `dynamic_rel` clause to deal with non-uniform duration of loop iterations. Ten iterations are folded within a single chunk of work. Dynamic scheduling of iterations is augmented with duty cycling.

We consider ten different degradation scenarios, namely ten aging index distributions, with worst-case degradation requiring up to 63% idleness. Results are then shown as

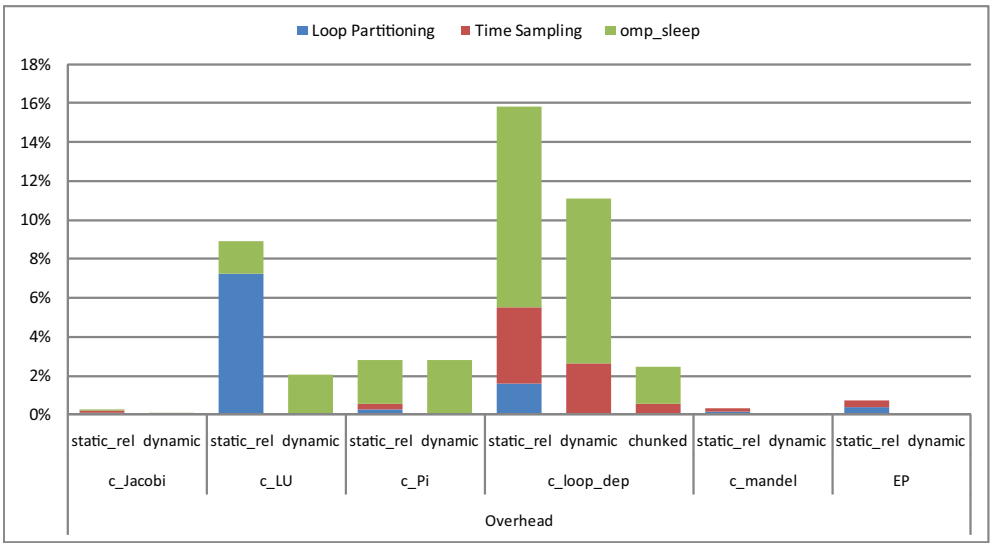


Figure 6.2: Sources of overhead

an average of several program runs under these platform degradation scenarios. In the following subsections we provide detailed information for each class of results.

6.5.1 Overhead

Figure 6.2 shows a breakdown of the considered sources of overhead, namely:

1. *Loop partitioning*: Time taken by the aging-aware partitioning algorithm to compute iteration spaces for every thread

2. *Time sampling*: Overhead due to loop body instrumentation for measuring the duration of iterations

3. *omp_sleep*: Overhead due to computation of idle time (based on profiled active time and aging index) in `omp_sleep` function

For benchmarks ***c_Jacobi***, ***c_Pi***, ***c_mandel*** and ***EP*** the overhead is always very small (under 3%). Parallelizing the main loop in benchmark ***c_LU*** with the `static_rel` clause brings a 9% overhead, which is mainly due to the execution time taken by the partitioning algorithm. This happens because this benchmark features a two-level loop nest, with the innermost nest being parallelized. Since this nest scans the rows of an upper-triangular matrix, decreasing amounts of work are scheduled with repeated

invocations to the partitioning algorithm. This overhead notwithstanding, we will show in Section 6.5.4 that our aging-aware `static_rel` clause achieves the best results in minimizing the performance loss.

Benchmark ***c_loop_dep*** is a synthetic benchmark in which a backward loop carried dependency is resolved through array replication. Each iteration in this program only contains a single write/read instruction in the array. Thus, in this benchmark all sources of overhead – which are usually negligible in real applications – are visible. When applying static aging-aware partitioning (`static_rel`) the biggest source of overhead ($\approx$ 10%) is the computation of the required sleep time for each core. Second for importance, is the overhead for profiling iteration execution time ($\approx$ 4%). Finally, loop partitioning accounts for an additional 1% overhead. Overall, the overhead introduced by our aging-tolerant facilities amounts to around 15% for static scheduling and 11% for dynamic scheduling. It is important to stress that the overhead is big because of the synthetic nature of the program and the poor computation performed in each iteration. When specifying a chunk size of 10 (i.e. folding ten iterations in a single task) we are able to reduce our techniques' overhead for the class of loops represented by the ***c_loop_dep*** benchmark to less than 3% (***chunked*** bar). In the following – if not differently specified – dynamic scheduling for ***c_loop_dep*** benchmark is chunked.

6.5.2 Idleness

Table 6.3 shows the results of rest time accuracy. Numbers in the table represent the percent error in distributing on each core the target amount of idle time. This error is caused by overhead code which is not managed by our aging-aware balancing policies. Since each parallel region may feature multiple loop nests, as well as code not contained within work-sharing constructs, we compute the actual core idleness as a percentage of the overall parallel region execution time to estimate the error. The results show

	c_Jacobi	c_LU	c_Pi	c_loop_dep	c_mandel	EP
static + sleep	-0,08	0,02	0,81	-2,84	0,01	-0,30
static_rel	-0,17	-3,98	0,74	-3,26	-0,09	-0,58
dynamic	-0,08	0,24	0,62	-2,32	0,00	-0,29
chunked	-	-	-	-0,65	-	-

Table 6.3: Percent error in target idleness distribution

that the error is always under 4%, thus confirming the effectiveness of the technique in offering idleness distribution precision. Benchmark ***c_LU*** has a very small error for ***static + sleep*** and ***dynamic*** configurations. The error is bigger when using static aging-aware scheduling (***static_rel***). This was expected, since as discussed in Section 6.5.1 the use of this clause carries an overhead that is not considered in duty cycling, thus leading to the error in idleness distribution. Similarly, for benchmark ***c_loop_dep*** sources of overhead present both in static and dynamic parallelization schemes lead to an error in idleness distribution accuracy that is greatly reduced when employing chunked scheduling.

6.5.3 Load balancing

As described in Section 6.4.1, we devised a partitioning algorithm that aims at keeping different threads workload as balanced as possible. In this section we provide results that confirm the effectiveness of the proposed approach.

Standard deviation of parallel execution time over cores has been normalized to the mean to provide a qualitative measure of the load imbalance. Results are shown in Figure 6.3 for different parallelization schemes. The black bars represent the original

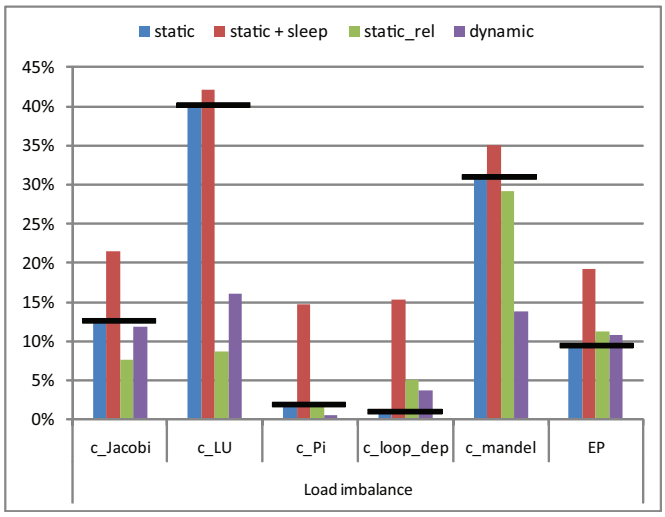


Figure 6.3: Load balancing

program parallelized with aging-agnostic OpenMP facilities, and thus is considered as a baseline. Looking at the black bars only, our set of benchmarks can be divided in

two categories. ***c_Jacobi***, ***EP***, ***c_Pi*** and ***c_loop_dep*** are regular and balanced. Each of them shows a deviation from average execution time which is contained within 13%.

On the other hand, ***c_LU*** and ***c_mandel*** have a degree of imbalance greater than 30%. ***c_mandel*** is known to have very unbalanced iterations, since decision on whether complex points belong to the Mandelbrot set area are taken within an inner loop which may take very different number of (inner) iterations to reach convergence. Similarly, LU decomposition has decreasing duration of inner loop iterations due to the diminishing number of elements in scanning an upper triangular matrix.

For all benchmarks, the naive ***static + sleep*** approach – which simply introduce idle times without re-allocating workload – un-surprisingly increase imbalance. Our partitioning algorithm (***static_rel***) is expected to never increase imbalance. This is confirmed by the plot. In cases like ***c_LU*** our algorithm reduces imbalance, since it schedules iterations in a smarter way. For example, when there are less iterations than cores, work is allocated to most reliable processors – which require smaller idle times – thus reducing the impact of duty cycling on load imbalance. Dynamic scheduling was originally meant to deal with balancing issues, so – as expected – even when augmented with aging-related features it preserves excellent balancing.

6.5.4 Performance Loss

As previously discussed, distributing idleness to degraded cores has a cost in terms of performance. Our partitioning algorithm aims at reducing this performance loss. According to the description given in section 6.4.1, part of the iterations originally assigned to degraded cores are re-distributed to more reliable cores. To estimate the effectiveness of this approach, we compare the parallel execution time of our aging-aware scheduling techniques against the naive ***static + sleep*** approach. The results are plot as a series of bars (for different program configurations) in Figure 6.4. We see that for all benchmarks except ***c_mandel***, our `static_rel` clause affords a significant reduction in performance loss w.r.t. ***static + sleep*** (around 50%). As explained in the previous section, for unbalanced applications such as ***c_mandel*** static scheduling schemes should be avoided. If iterations are known to have different duration, evenly dividing the iteration space among cores results in unbalanced execution time. The same clearly applies to our aging-aware partitioning algorithm, so we did not expect `static_rel` to provide good results. It is important here to stress that this is NOT

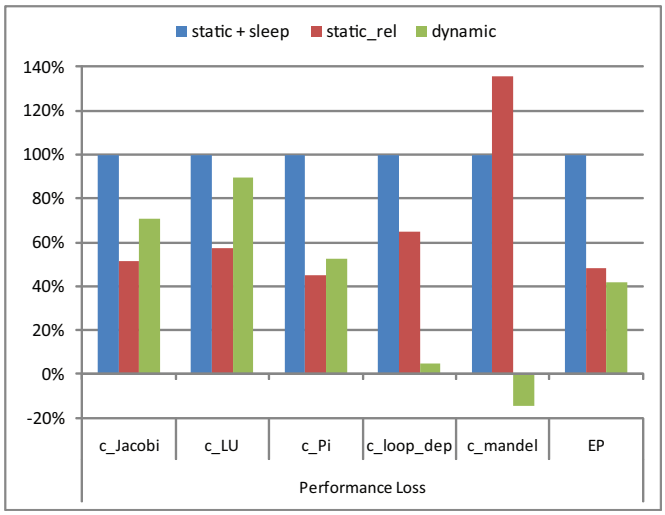


Figure 6.4: Performance loss reduction

a problem of our partitioning scheme, but rather an inherent limitation of static parallelization. The knowledgeable programmer would rather employ dynamic scheduling to deal with similar scenarios. For this benchmark in particular, employing a dynamic scheduling policy achieves better performance results than the original static scheduling notwithstanding the idle periods. Finally, we can notice that there is a big difference in performance loss reduction between static and dynamic aging-aware scheduling for benchmark ***c_loop_dep*** (red and green bars). This is justified by the fact that dynamic scheduling here employs chunking, thus reducing the sources of overhead described in Section 6.5.1.

6.6 Conclusion

In this chapter we described a compiler-supported technique for NBTI-induced aging tolerance in data-intensive MPSoCs. The technique is able to finely insert periods of recovery to various cores within loop executions to compensate non-homogeneous core degradation and minimize the performance impact through loop iteration reallocation among cores.

This technique has been implemented as a set of extensions to the OpenMP parallel programming model. Experimental results on a distributed shared memory multiprocessor platform demonstrate its accuracy in idleness allocation and performance impact

minimization.

Much work remains to be done to accurately model the effect of idle periods on degraded cores. The aging models in our simulator must be enhanced to actually provide a feedback on the entity of the recovery induced by the applied idleness. This will allow to revisit and extend the aging-tolerant techniques in two main directions:

1. Currently the duration of idle periods is decided based on the profiled execution time of a chunk of iterations. This clearly also includes time spent on memory or I/O, which may not be actually impacting core degradation.

2. Currently the techniques are focused on loop-level parallelism (`pragma omp for`), whereas at the moment for task-level parallelism (`pragma omp sections`) we only allow duty cycling at a coarse task-wide granularity.

Bibliography

[1] A. Marongiu, and L. Benini. Efficient OpenMP support and extensions for MPSoCs with Explicitly managed memory hierarchy. *DATE '09: Proceedings of the 12th International Conference on Design, Automation and Test in Europe*, pages 809–814, 20-24 Apr. 2009.

[2] M. Agarwal, B. Paul, M. Zhang, and S. Mitra. Circuit failure prediction and its application to transistor aging. *Proceedings of the 25th IEEE VLSI Test Symposium table of contents*, pages 277–286, May 2007. 116, 119, 121

[3] C. d. S. F. Dorta, and A.J. Rodriguez. The OpenMP source code repository. *Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on*, pages 244– 250, 9-11 Feb. 2005. 128

[4] M. Eireiner, S. Henzler, G. Georgakos, J. Berthold, and D. Schmitt-Landsiedel. In-situ delay characterization and local supply voltage adjustment for compensation of local parametric variations. *Solid-State Circuits, IEEE Journal of*, 42(7):1583–1592, July 2007. 116, 119

[5] S. Hong, S. Narayanan, M. Kandemir, and O.Ozturk. Process variation aware thread mapping for chip multiprocessors. In *DATE '09: Proceedings of the 12th International Conference on Design, Automation and Test in Europe*, pages 821–826, 20-24 Apr. 2009. 119

[6] http://www.nas.nasa.gov/Resources/Software/npb.html. Nas parallel benchmarks. 128

[7] W.-C. Jeun, and S. Ha. Effective OpenMP implementation and translation for multiprocessor system-on-chip without using OS. *Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific*, pages 44–49, 23-26 Jan. 2007.

[8] K. Kang, S. P. Park, K. Roy, and M. A. Alam. Estimation of statistical variation in temporal NBTI degradation and its impact on lifetime circuit performance. In *ICCAD '07: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 730–734, Piscataway, NJ, USA, 2007. IEEE Press. 115, 116

[9] E. Karl, D. Blaauw, D. Sylvester, and T. Mudge. Multi-mechanism reliability modeling and management in dynamic systems. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(4):476–487, April 2008. 115

[10] A. Krishnan, V. Reddy, S. Chakravarthi, J. Rodriguez, S. John, and S. Krishnan. NBTI impact on transistor and circuit: models, mechanisms and scaling effects [mosfets]. *Electron Devices Meeting, 2003. IEDM '03 Technical Digest. IEEE International*, pages 14.5.1–14.5.4, Dec. 2003. 116

[11] S. V. Kumar, C. H. Kim, and S. S. Sapatnekar. An analytical model for negative bias temperature instability. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 493–496, New York, NY, USA, 2006. ACM. 116

[12] F. Liu, and V. Chaudhary. Extending OpenMP for heterogeneous chip multiprocessors. *Parallel Processing, 2003. Proceedings. 2003 International Conference on*, pages 161–168, 6-9 Oct. 2003.

[13] F. Liu, and V. Chaudhary. A practical OpenMP compiler for system on chips. *International Workshop on OpenMP Applications and Tools, WOMPAT 2003*, pages 54–68, June 2003.

[14] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang. Supporting OpenMP on cell. In *IWOMP '07: Proceedings of the 3rd international workshop on OpenMP*, pages 65–76, Berlin, Heidelberg, 2008. Springer-Verlag.

[15] D. Roberts, R. G. Dreslinski, E. Karl, T. Mudge, D. Sylvester, and D. Blaauw. When homogeneous becomes heterogeneous. In *Third workshop on Operating Systems for Heterogeneous Multiprocessor Architectures (OSHMA)*, Sep. 2007. 118

[16] A. Tiwari and J. Torrellas. Facelift: Hiding and slowing down aging in multicores. In *41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 129–140, 2008. 116, 118, 120

[17] F. Wang, C. Nicopoulos, X. Wu, Y. Xie, and N. Vijaykrishnan. Variation-aware task allocation and scheduling for MPSoC. In *ICCAD '07: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design*, pages 598–603, Piscataway, NJ, USA, 2007. IEEE Press. 118

[18] J. Winter, and D. Albonesi. Scheduling algorithms for unpredictably heterogeneous CMP architectures. In *38th International Conference on Dependable Systems and Networks*, pages 42–51, 24-27 Jun. 2008. 118

Chapter 7

Publications

A. Marongiu, L. Benini and M.T. Kandemir, *Lightweight barrier-based parallelization support for non-cache-coherent MPSoC platforms*, CASES 2007, pag.145-149

A. Marongiu, L. Benini, A. Acquaviva and A. Bartolini, *Analysis of Power Management Strategies for a Large-Scale SoC Platform in 65nm Technology*, DSD 2008, pag.259-266

M. Schindewolf, A. Cohen, W. Karl, A. Marongiu and L. Benini, *Towards Transactional Memory Support for GCC*, GROW 2009

A. Marongiu and L. Benini, *Efficient OpenMP Support and Extensions for MPSoCs with Explicitly Managed Memory Hierarchy*, DATE 2009, 809-814

A. Marongiu, A. Acquaviva and L. Benini, *OpenMP Support for NBTI-Induced Aging Tolerance in MPSoCs*, SSS 2009, pag.547-562

A. Marongiu, M. Ruggiero and L. Benini, *Efficient OpenMP Data Mapping for Multicore Platforms with Vertically Stacked Memory*, DATE 2010

A. Marongiu and L. Benini, *An OpenMP Compiler for Efficient Array Partitioning in Distributed Shared Memory MPSoCs*, ***Under review*** – IEEE Transactions on

Computers

A. Marongiu, P.Burgio and L. Benini, *Evaluating OpenMP Support Costs on MP-SoCs*, ***Under review*** – Euromicro DSD 2010

S. Raghav, M. Ruggiero, D. Atienza, C. Pinto, A.Marongiu and L. Benini, *Scalable Instruction Set Simulator for Thousand-core Architectures Running on GPGPUs*, ***Under review*** – WEHA 2010

Chapter 8

Conclusion

The efficient use of a machine's memory system and parallel processing resources has become one of the most important challenges in program optimization for embedded MPSoCs. Moreover, efficient use of the memory hierarchy is increasingly important because of the power cost of data access through the program. Architecture trends are leading to large scale parallelism using simpler cores and progressively deeper and complex memory hierarchies. These new architecture designs have improved power characteristics and can offer large increases in performance, but traditional programming techniques are inadequate for these architectures.

In this dissertation, we explored programming features and runtime support for making efficient use of the memory hierarchy. More specifically, we extend the programming API of OpenMP with custom directives and clauses that allow to identify candidate arrays in a program for partitioning. Partitioned arrays are distributed among the memory hierarchy so as to maximize the number of each processor's references that are satisfied from its local SPM. We evaluated the applicability of this enhanced OpenMP programming framework on generic and representative embedded PGAS MPSoC templates. We also provide preliminary results and experience with adapting the array partitioning techniques to MPSoCs with vertically stacked DRAM memory.

Finally, we describe how our enhanced OpenMP programming framework can further be extended to deal with different sources of heterogeneity in MPSoCs, namely non-uniform processing resources due to core aging effects. We present techniques which leverage a partial recovery effect inherent in the considered aging phenomenon,

Negative Bias Temperature Instability (NBTI), to schedule work to processor so as to maximize system lifetime.