Università di Pisa

Dipartimento di Informatica
Dottorato di Ricerca in Informatica

Ph.D. Thesis

Optimizations and Cost Models for multi-core architectures: an approach based on parallel paradigms

Daniele Buono

Supervisor
Marco Vanneschi

May 13, 2014

Dipartimento di Informatica, Largo B. Pontecorvo, 3, I-56127 Pisa, Italy
SETTORE SCIENTIFICO DISCIPLINARE INF/01

Abstract

The trend in modern microprocessor architectures is clear: multi-core chips are here to stay, and researchers expect multiprocessors with 128 to 1024 cores on a chip in some years. Yet the software community is slowly taking the path towards parallel programming: while some works target multi-cores, these are usually inherited from the previous tools for SMP architectures, and rarely exploit specific characteristics of multi-cores. But most important, current tools have no facilities to guarantee performance or portability among architectures. Our research group was one of the first to propose the structured parallel programming approach to solve the problem of performance *portability* and *predictability*. This has been successfully demonstrated years ago for distributed and shared memory multiprocessors, and we strongly believe that the same should be applied to multi-core architectures.

The main problem with performance portability is that optimizations are effective only under specific conditions, making them dependent on *both* the specific program and the target architecture. For this reason in current parallel programming (in general, but especially with multi-cores) optimizations usually follows a try-and-decide approach: each one must be implemented and tested on the specific parallel program to understand its benefits. If we want to make a step forward and really achieve some form of performance portability, we require some kind of prediction of the expected performance of a program. The concept of *performance modeling* is quite old in the world of parallel programming; yet, in the last years, this kind of research saw small improvements: cost models to describe multi-cores are missing, mainly because of the increasing complexity of microarchitectures and the poor knowledge of specific implementation details of current processors.

In the first part of this thesis we prove that the way of performance modeling is still feasible, by studying the Tilera Tile*Pro*64. The high number of cores on-chip in this processor (64) required the use of several innovative solutions, such as a complex interconnection network and the use of multiple memory interfaces per chip. For these features the Tile*Pro*64 can be considered an insight of what to expect in future multi-core processors. The availability of a cycle-accurate simulator and an extensive documentation allowed us to model the architecture, and in particular its memory subsystem, at the accuracy level required to compare optimizations.

In the second part, focused on optimizations, we cover one of the most important issue of multi-core architectures: the memory subsystem. In this area multi-core

strongly differs in their structure w.r.t off-chip parallel architectures, both SMP and NUMA, thus opening new opportunities. In detail, we investigate the problem of data distribution over the memory controllers in several commercial multi-cores, and the efficient use of the cache coherency mechanisms offered by the Tile*Pro*64 processor.

Finally, by using the performance model, we study different implementations, derived from the previous optimizations, of a simple test-case application. We are able to predict the best version using only profiled data from a sequential execution. The accuracy of the model has been verified by experimentally comparing the implementations on the real architecture, giving results within $1 - 2\%$ of accuracy.

A zia Isa, nonna Franca e zio Gianfranco,
perché mi hanno donato un sogno.

E alla mia famiglia,
perché mi ha aiutato a realizzarlo.

Computer Science is a science of abstraction - creating the right model for a problem and devising the appropriate mechanizable techniques to solve it.

Foundations of Computer Science,
A. Aho and J. Ullman

Acknowledgments

Questa è la storia di un sogno. Un sogno iniziato il giorno della mia prima comunione, quando zia, zio e nonna mi regalarono il mio primo computer. Un sogno che ho custodito e nutrito per tutti questi anni e che ora, posso dire con enorme gioia, si sta realizzando.

In un'avventura così lunga ho incontrato tantissime persone, ed è grazie a tutte loro che oggi sono qui, a culminare il sogno di un bambino. Vorrei ringraziarvi tutti, a partire dai professori che dalle medie in poi mi hanno aiutato in questo percorso: Maffei, Fabretti, Carosi, Uggeri, Adriani, Sajeva, Tore, Dore, e tutti quelli che mi sono dimenticato.

Un grazie di cuore al Prof. Marco Vanneschi, supervisore non solo di tesi, ma di tutto il mio percorso: le tue lezioni di architettura rimarranno per sempre nel mio cuore, con quel misto di ammirazione... e di paura!

Grazie anche al Prof. Marco Danelutto, per tutto il supporto che mi ha dato in questi anni, ma soprattutto... per l'articolo mandato ad Amsterdam!

Un ringraziamento speciale va sicuramente a Gabriele, compagno di ufficio e di articoli, ma soprattutto spalla su cui piangere nei momenti no di questo dottorato (e ce ne sono stati tanti!). Grazie a Massimo, ai mille caffè (letteralmente) presi insieme e alle annesse chiaccherate sulla ricerca. Grazie ad Alessio, per gli amici PasQ, il famoso kernel hacker; nonostante le nostre strade si siano divise, rivederci è sempre un piacere.

Grazie a tutti i componenti, passati e futuri, del gruppo: Silvia, Tixi, Fabio, Paolo, Daniele, Massimiliano e Carlo. Grazie agli indimenticabili sfidanti: Giacomo, Andrea e Sabri.

Grazie al mio mitico gruppo del corso di laurea: Bacai, Profe, Sandro, Gabri, Lotta e Cino. Anche se ormai ognuno di noi ha preso la sua strada, dobbiamo continuare a vederci ogni tanto per giocare a Risiko e/o Dominion!

Un ringraziamento speciale anche al badge del dipartimento, che mi ha permesso di entrare agli orari più impensabili.

Grazie ovviamente alla mia famiglia, che nonostante tutto mi segue e mi supporta (o meglio, sopporta) ormai da 29 anni. Mamma, babbo, Jessy, Nerone e tutti i nonni: se sono oggi qui è soprattutto merito vostro!

Grazie anche a Serena, non più compagna di vita ma sicuramente amica unica e indimenticabile.

Infine, grazie a tutti quelli che non ho menzionato: vi porto comunque nel cuore.

Contents

List of Figures

List of Tables

Part I

Introduction

Chapter 1

Introduction

The trend in modern microprocessor architectures is clear: multi-core chips are here to stay. Current processors are composed of 4 to 12 cores on the same chip, and this number is continuously increasing every year, up to the point that the term many-cores have been introduced, to indicate the large amount of core per chip of some solutions. At the same time these processors usually implements simultaneous multi threading (SMT[166]), allowing the execution of up to 4 threads on the same core, and server configurations provides multiple processors on the same board, giving even more cores on a single machine. Some notable examples of current highly parallel multi-core platforms are the Tilera Tile*Pro*64 [26] (64 cores), the future IBM PowerEN[83] (16 4-way SMT cores each processor for a maximum of 64 cores and 256 thread in 4-chip configurations), the AMD Opteron[62] with 48 cores in a single machines, the IBM Power7[103] servers, supporting up to 32 8-core 4-way SMT processors per machine, for a total of 1024 threads or the Intel Xeon Phi, an accelerator composed of 60 4-way SMT cores[98]. Current research works for future architectures expect Chip MultiProcessors with 128[148] to 1024[110] cores on a chip in some years.

In spite of this, the software community is slowly taking the path towards parallel programming. And many scientists believe it is taking the *wrong* path[15].

A wide range of parallel programming tools target multi-cores; yet these are usually inherited from the previous tools for SMP architectures, and rarely exploit specific characteristics of multi-cores.

But most important, current tools have no facilities to guarantee performance or portability among architectures. Unfortunately this have deep implications in the development of a parallel program, because you cannot have even a rough idea of the performance of your program until it is run on a *specific* platform, thus making different parallelizations of a program incomparable in a formal or generalizable way, but only by execution times.

Our research group proposed the structured parallel programming approach for distributed and shared memory multiprocessor architectures some years ago[172], to solve the performance *portability* and *predictability* problem, as well to speed up

parallel program development; we strongly believe that the same approach can (and must) be applied to multi-core architectures.

Multi-cores can be considered shared memory multiprocessors integrated in a single chip: indeed they are also called Chip MultiProcessor (CMP) in academic and research world. General results in parallel programming for multiprocessors are therefore valid for CMPs; at the same time, however, there are some important features that are not available in multiprocessor and must be further investigated.

Just like in multiprocessors, many architectural choices are possible when building CMPs: the number and complexity of cores, the interconnection network among cores and towards the outer memory, cache hierarchies and cache coherence protocols. However, given the limited chip size, engineers have to find a trade-off between the features of each component, limiting the wide range of possibilities usually available in common multiprocessors.

At the same time the integration on a single chip allows processing cores to share resources: nowadays cores usually share a cache level[62, 83] (typically L2 or L3) to offer faster communications and to better exploit caches, but in some cases even functional units (for example the floating point unit of the UltraSPARC-T1[107] and the newest AMD Opteron processors[51]) are shared among cores for power consumption and chip complexity reasons. Moreover the idea of fetching instructions from different control flows emerged in the last decade, to better use the processor functional units, hide load/store latencies and overcome the limits of instruction-level parallelism: today SMT is available in almost every high-end processor, including CMPs[103, 107].

1.1 Structured parallel programming

All these degrees of freedom make each multi-core different, forcing the programmer to write specific low-level code to reach good (and only hopefully the best achievable) performance. In fact, since its introduction, parallel programming was strictly related to HPC environments, in which programmers were usually willing to write parallel code by mean of low level libraries that, giving a complete control over the parallel application, allowed them to manually optimize the code and exploit at best the architecture. However, this programming methodology exposed its first problems with the emergence of cluster and grid computing, when the parallel architectures become dynamic and heterogeneous, therefore limiting the possibilities of ad-hoc optimizations. With multi-cores, the massive introduction of parallel architectures in every device, and thus in every computing sector, critically exposed the lack of proper tools to easily implement a parallel application: the industry cannot afford the cost of re-writing (or even re-tuning) an application for every available computing architecture.

As common in computer science, we believe that the answer is to *abstract* the problem, and work at a higher level. Consistently with this approach, the idea of *automatic parallelization* has been investigated in the past. However, despite the increasing effort (especially of the scientific environment), automatic parallelizers are still ineffective in fully exploiting parallel architectures, rarely providing good speedups even on 4 and 8-cores [52, 82]. This could have been considered sufficient at the introduction of multi-cores when we had 1 to 4 processors per chip; after almost a decade of chip multiprocessors, however, the current number of cores require a more scalable approach.

To the current state of the art, in short, a rewrite of the program is needed to exploit parallelism. In this case, a proper mix of *ease of use* and *performance* is still the main concern of researchers. It is widely acknowledged by the scientific literature [69, 171, 152] that performance portability is achievable only by using a high-level approach to parallel programming: exactly as in sequential programs, where portability is guaranteed by sequential high-level languages, we need to define parallel constructs that allow a proper compiler to produce efficient code for any architecture.

In other terms, by using a high-level parallel programming model, we should be able to describe our parallel application and be sure that it will perform reasonably well on the wide choice of parallel architectures available today.

At this time, some works for parallel programming on multi-cores exist; however these are usually shared memory multiprocessor programming tools that flatten the differences among cores on the same and on different chips. These tools are usually given as complete programming languages or libraries:

- **Programming languages**: some examples are *Chapel*[53], *Berkeley Unified Parallel C*[77], *Cilk*[100], OmpSs[76], OpenCL[155] or *OpenMP*[66]; they are entirely new programming languages or extensions for existing sequential languages.

- **Libraries**: *Intel Threading Building Blocks*[142], *Skandium*[114], *FastFlow*[12], *Intel Array Building Blocks*, SkelCL[154] and many others.

In general, all these tools express some characteristics of high-level parallel programming, i.e. help the programmer by easing the burden of writing parallel applications. For example, OpenMP uses a shared-memory programming model, but allow the programmer to extend sequential code by mean of annotations, without explicitly writing the parallel threads. However, the programmer must know basic concepts of the resulting parallelization to ensure program correctness. Furthermore, in many cases a detailed knowledge of the parallel implementation and proper annotations or code reorganizations are required to ensure good parallel performance[109].

The same concept apply, more or less, to all the discussed languages and libraries: they help the programmer in many ways, especially hiding details typical of low-level parallel programming, but they do not allow that kind of *performance portability* described before.

Among the set, FastFlow, Skandium and SkelCL are probably the higher level tools currently available, as they represent the class of *Structured Parallel Programming models*.

We consider structured parallel programming the most interesting class of high-level parallel models. It started with the concept of algorithmic skeletons by Cole [61] and has been successfully applied basically in any possible parallel environment, starting from clusters[67] and shared memory machines[114], to grid[7], cloud and pervasive environments[32].

The main idea behind structured parallel programming is to let the programmer define an application by means of *parallel patterns* (also called *paradigms*). A parallel pattern describe, in a general way, the structure of the interactions of a parametric set of entities. With parallel paradigms, the programmer just select the proper pattern and describe the sequential code to be inserted in the entities of the pattern. The rest of the code is produced by the programming environment.

1.2 Parallel patterns and their optimizations

A parallel paradigm describes the abstract parallel entities and the structure of the interactions. However, the structure given by the paradigm is very general, so that there are many ways of coding it on a parallel machine. We usually find a simple implementation that strictly resemble the definition of the parallel paradigm, but there are many different versions that may perform better than the baseline, depending both on the algorithm and on the deployment architecture. For example, even on simple paradigms such as the task farm, we can find different approaches, such as hierarchic scheduling, master/worker or emitter/worker/collector schemes, load balancing techniques, different cooperation mechanisms and much more [28, 143].

On top of this, we can find some "Pattern-Independent Optimizations", i.e. optimizations that are applicable to parallel programs in general. Of course, applicability does not always result in performance improvements: some pattern will probably benefit more than others of the single optimizations. In general, however, a pattern-independent optimization:

a) **is independent**, as it does not need a specific pattern to be applied;

b) **deeply affect the implementation**, so that it may significantly drive the pattern implementation towards specific forms, and therefore

c) **affect minor pattern-specific optimizations**, that could both be not be feasible or ineffective.

We therefore consider these the *starting point* towards efficient pattern implementations.

Keeping in mind the long-term research objective of our group, with this thesis we will address two of this kind of optimizations **specific of multi-core architectures**. In particular, we will focus on techniques to exploit the memory hierarchy of these processors, which is sensibly different w.r.t. other kind of multiprocessor architectures.

1.2.1 Multiple memory interfaces

The increase of the number of core inside the chip is exacerbating the memory wall[16] problem. To solve this, apart from sensibly increasing the amount of on-chip caches, an increasing number of architectures is encapsulating multiple memory interfaces on-chip. Many chip manufacturer also allows the composition of a limited number (2-4) of chip per computer, further increasing the total number of memory controllers. This way of increasing the total amount of memory bandwidth, however, is starting to pose problems to the programming model: indeed these architectures are not UMA (Uniform Memory Architecture) anymore, but are neither as those NUMA (Non-Uniform Memory Architecture) architectures studied in the past years. Current research[39, 70] is attacking this problem from the operating-system point of view (i.e. how to better allocate virtual memory pages among the many memory controllers), therefore not with a parallel programming vision of the problem.

With this thesis we will study *how* a parallel paradigm should be implemented to exploit at best the multiple memory controllers, starting with the solution already studied in the past for SMP and NUMA architectures, that will be adapted and extended to better fit this kind of multiprocessors.

1.2.2 Automatic Cache Coherence

It is today acknowledged that shared-memory parallel architectures should provide some cache coherence facility to ensure parallel correctness.

In general shared memory programming models an automatic cache coherence protocol is proven to offer the best performance[138]. There are, however, several works that highlight how cache coherence *may* be ensured at a software-level to obtain performance improvements[1]. This has generally been a slippery ground, because current architectures *do not* usually allow disabling automatic cache coherence, so basically any claim has been proved by mean of architecture simulations. The emergence of many-core architectures somewhat changed the scenario: handling cache coherence among a large set of processors is indeed an expensive operations,

so that some architectures, such as the Tilera Tile*Pro*64, allow to *control* and *disable* the automatic cache coherence facilities. This opens new and exciting research aspects, especially when mixed with the Structured Parallel Programming, where software cache-coherence *should* be efficiently implemented at the support level with absolute transparency with respect to the application programmer. We will approach this problem by examples, showing how, in particular cases, the knowledge given by the parallel pattern is *sufficient* to guarantee correctness with incoherent memory areas, and may also provide better performances w.r.t. the corresponding program run with automatic cache coherence.

1.3 Introducing a performance model

The main problem of having multiple implementations is that there does not usually exist the "best" that outperforms the others; in general, it is the combination of the architecture, the parallel pattern and the program that determines the best implementation: for example, specific communication and synchronization mechanisms may benefit more than others of the underlying cache coherence mechanisms and/or interconnection network; this advantage can be important or negligible for a specific application, depending on the parallel pattern and/or the coarse/fine grain of the program.

Furthermore, we should consider that, in general, a parallel application is composed of several patterns that coexist on the same architecture and cooperate to compute the final result. In such environment, a careful resource allocation is required to obtain the best performance for the overall program, which *may* not directly imply the best performance for each pattern. In this scenario a performance model that correctly estimate the performance of each pattern is required to optimize the whole application.

Performance prediction of a program, although widely studied, is still an open problem in the research community. Cost models in the world of parallel programming are usually proposed to asymptotically study algorithms, like PRAM[80], BSP[169] and Multi-BSP[170] models. A first step towards a more "detailed" model is LogP[63], and its successive enhancements. However all these models are kept as simple as possible to let programmers easily compare algorithms. We are not interested in this kind of models. We are looking for a more realistic model that takes into account every *important* property of the parallel architecture and of the parallel program. The model does not have to be simple, because it will not be used by the programmers, but by the parallel programming framework. Unfortunately, as of today, there does not exist a way to precisely estimate the completion time of a general program on current architectures, mainly because of their complexity and dynamicity.

The idea of specific performance models for parallel pattern is not really new, as it was introduced in P^3L[18] to select the so-called "implementation templates".

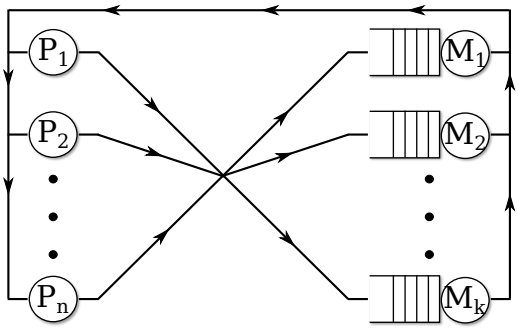


Figure 1.1: Bhandarkar's queueing network to model the memory contention.

These, however, modeled the performance of the implementations by taking the sequential code as a "black box", with specific, immutable, characteristics. We extend the original concepts by introducing an *architecture model*, that allow us to predict the performance degradation that occur when multiple processes are executed on the same multi-core chip. It is widely known that the major source of degradation is the sharing of the memory subsystem. One of the first to analyze this form of degradation has been Bhandarkar[36], that modeled the processor-memory subsystem with the queueing network in Figure 1.1, where each processor generate a memory request and then stop its execution waiting for the response. We will start from this simple yet effective model, and adapt it for a current multi-core architecture: the Tilera Tile*Pro*64.

By using the obtained model, parameterized for specific programs, we will be able to estimate the average response time of the memory controller, an indispensable value to accurately predict the performance of the parallel program on a shared memory architecture. Although useful per se, to predict the response time for a generic parallel program, this model will become the cornerstone in our approach, to study and compare the different implementations of each pattern and the pattern-independent optimizations.

1.4 Towards a parallel programming environment

Our research group history in structured parallel programming is quite long, starting with the P^3L skeleton language in 1992, and culminating with ASSIST in the last years. We never, however, really focused our efforts in multi-core and shared memory architectures in general. Our experiments with FastFlow[9] demonstrated the need, and the possibility, of multi-core-specific optimizations in a skeleton-based library. A skeleton library, however, does not allow us to fully exploit the benefits of structured parallel programming, because it does not (entirely) allows code restructuring and transformations. With this thesis we address the problem from a different point of

view, laying the foundations for a complex environment capable of automatic code rewriting and optimizations for this class of architectures.

The long-term project of our research group is ASSISTANT, the extension and adaptation of ASSIST for the current world of parallel computing, composed of multi-cores, pervasive grids and clouds. Many of the principles introduced in ASSIST are *inherited* and *extended*, in order to provide a significant leap forward in the world of multi-core-oriented parallel programming.

Respecting the basic ASSIST principles, a parallel program will be described as a generic graph of stream-connected parallel modules. Each module will be constituted by a parallel pattern, and the programmer will be able to write the algorithm code by mean of the most used sequential languages (C, C++, Matlab, Java, and so on).

As already mentioned, programming models based on libraries are considered unsuitable for achieving the desired level of programmability and performance portability: our environment will need an intelligent *source-to-source* parallel compiler, able to analyze the module-based description to determine the possible parallel implementations, evaluate them for the target machine and, finally, produce the source code of a low-level parallel program.

Our past experience in parallel programming also pointed that there are many cases in which performance portability is not *completely* achievable at compile-time: the cost model may be not detailed enough to accurately fit the <application, implementation, architecture> tuple, or some model parameters may be unpredictable (because of both the architecture and the algorithm) so that a mere compiler-time performance portability becomes ineffective. To handle all these important cases, it is also mandatory to support *adaptivity*, by means of efficient run-time reconfigurations, in addition to static optimizations[32].

The resulting "compilation workflow" is depicted in Figure 1.2. Of course, the meaning of compilation now is stretched to the whole execution because of the run-time-based reconfigurations. We can easily notice how important is the *Cost Model*, that affects basically every step of the work-flow, making it a first-class citizen in our approach. In short, starting from the specification, we first use the cost model to statically derive a good parallelization of each module, selecting it from the possible implementations and optimizations of the pattern given by the programmer. Then, at run-time, we will continuously monitor the program, and re-apply the cost model to find, if possible, even better solutions.

1.5 List of Contributions of the Thesis

The road towards ASSISTANT is still long; with this thesis we start targeting multi-core architectures, showing the feasibility of the cost model approach, by defining the architectural model for a specific many-core architecture (the Tilera Tile*Pro*64), and applying it on well known parallel pattern implementations to evaluate specific memory-related optimizations introduced in the thesis.

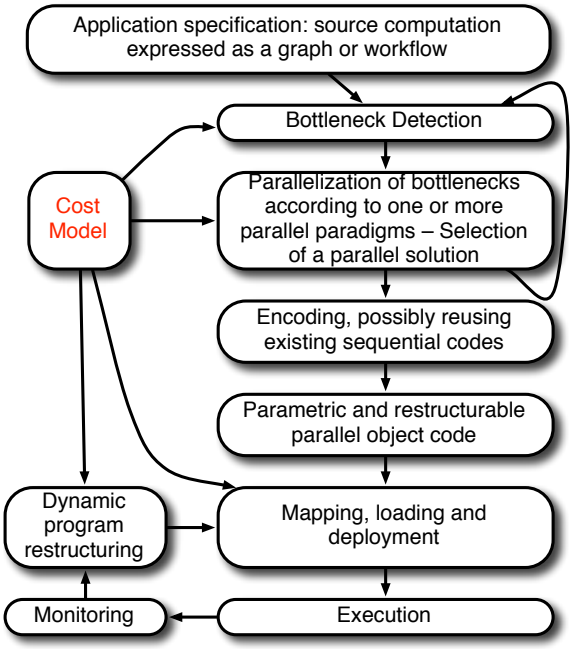


Figure 1.2: The "compilation workflow" in our programming environment.

The fundamental contributions are the following:

- An extensive study on how to effectively exploit the multiple memory interfaces available on current chip multiprocessors, that exhibit different characteristics w.r.t old-style UMA and NUMA architectures and thus may require a different approach. The results shows that, in general, an approach similar to NUMA architectures, in which we favor the use of a local environment for each process, allocated to the nearest memory controller, tends to perform slightly better than the others.

- An introductory study to software-managed cache coherence for parallel pattern implementations and its beneficial effects on the performance of the applications on the Tile*Pro*64 architecture.

- A queueing network model for the Tile*Pro*64 architecture, to accurately evaluate the performance effects of the sharing of the memory system on the memory response time. The accuracy of the model has been tested by using a cycle-accurate simulator of the architecture and benchmarks executed on the real platform, resulting in errors always below the threshold of 20% and, on average, of $\sim 10\%$.
  The generality of this model, that allow the definition of different behaviors for each processor of the system, makes it feasible to model basically any parallel program to be run on the architecture, and extract an average memory

response time.
In scenario of the structured parallel programming, the model can be automatically parameterized to evaluate different pattern implementation and of course, the optimization techniques emerged in the previously listed studies.

- A demonstration of the use of the performance model, to study different implementations of a parallel module on the Tile*Pro*64, modeling their service time to select, depending on the parallelism degree, the best implementation available. The estimations on the service times were compared with the real implementation, resulting in an approximation within the $\sim 2.5\%$ of error, resulting in the selection of the best solution in most cases.

A notable byproduct of the thesis was the **EQNSim** simulator, that consist in a simulation environment that allow us to simulate elements of queueing theory (i.e. queues) and mix them with more complex, user defined modules. The environment has been extensively used during the development of the architectural performance model of the Tile*Pro*64 to test the behavior of partial queueing network models inside complex environments. The simulator has thus been used throughout the rest of the thesis to solve the architectural model. Moreover, it has been actively used in our group for other researches; in particular, in [127, 126, 128] for the simulation of the behavior of a parallel adaptive program using different reconfiguration policies.

1.6 Outline of the Thesis

The thesis is conceptually organized in four separate parts:

Part I: Introduction in which we introduce the reader to the world of structured parallel programming and to the need of performance models, putting the basis to understand the following parts. In particular, we have:

- **Chapter 2** that review the current state of parallel programming for multicores. We describe the features of current architectures and the evolution trend that is likely to be followed. Then, a brief overview of the current tools for programming these processors is given, trying to highlight the problems of the current generation of parallel environments. Finally, we introduce the most common models used to evaluate the performance of parallel programs.

- **Chapter 3** motivates the use of high level parallel programming to achieve the level of portability required to minimize the efforts of developing parallel programs. We introduce the conceptual framework of ASSIST and its pervasive evolution ASSISTANT, that represents the long-term project of our research groups, and for which this thesis represent a small, yet very important tile. At the end of the chapter we also introduce the target architectures that will be used throughout the thesis to study and verify our ideas.

Part II: Cost Models which is probably the most innovative part of the thesis, as we introduce the general approach of ASSISTANT to performance models. In particular:

- **Chapter 4** describe the methodology we will use to estimate the performance of a parallel program, that is divided into: a) the use of an architectural model to predict the performance degradations related to the shared memory subsystem of multi-cores; b) the use of pattern-specific models to predict the performance of each parallel pattern implementation and c) the use of a generic methodology to evaluate the performance of the graph of modules that compose the parallel application.
- **Chapter 5** develop the model for a specific commercial architecture: the Tilera Tile*Pro*64, that constitute an interesting example of a chip multiprocessor, given its 64 cores and the use of innovative solutions for the interconnection network, the cache coherence mechanisms and the use of multiple independent memory interfaces on chip.

Part III: Optimizations where we analyze some optimization ideas especially targeted to multi-core architectures. In this part we try to maintain generality by not focusing on a specific architecture.

- **Chapter 6** deeply analyze the problem of having multiple memory interfaces on the same parallel machine. While this may seem an old problem, it is important to notice that in this area chip multiprocessor differs from both SMP and NUMA architecture, and how to exploit them at best has not yet approached systematically by the research world.
- **Chapter 7** introduce to the problem of automatically handling cache coherence on a large multiprocessor, and to the fact that, in some cases, the use of a software-defined cache coherence mechanism may improve the performance of the parallel application. We prove the idea by showing some preliminary tests on the Tile*Pro*64 architecture, in which we are able to disable automatic cache coherence.

Part IV: Wrapping Up conclude the thesis, by joining the results of Parts II and III into the evaluation of several implementation choice for a module.

- **Chapter 8** introduce a parallel module and evaluate, using the models of Part II, three different implementations of a farm pattern on the Tile*Pro*64 architecture. The implementations extensively use, among the others, the optimizations of Part III. The results are validated with the execution of these on the real Tile*Pro*64.
- **Chapter 9** present the conclusions of the thesis and the possible path towards an integration of these results in ASSISTANT.

1.7 Current publications by the author

The following represents the publications that I worked on during my Ph.D. research:

- C. Bertolli, D. Buono, G. Mencagli, and M. Vanneschi. Expressing adaptivity and context-awareness in the assistant programming model. In *Proceedings of the Third International ICST Conference on Autonomic Computing and Communication Systems*, 2009

- C. Bertolli, D. Buono, S. Lametti, G. Mencagli, M. Meneghin, A. Pascucci, and M. Vanneschi. A programming model for high-performance adaptive applications on pervasive mobile grids. In *Proceeding of the 21st IASTED International Conference on Parallel and Distributed Computing and Systems*, 2009

- C. Bertolli, D. Buono, G. Mencagli, M. Mordacchini, F. M. Nardini, M. Torquati, and M. Vanneschi. Resource discovery support for time-critical adaptive applications. In *The 6th International Wireless Communications and Mobile Computing Conference. Workshop on Emergency Management: Communication and Computing Platforms*, 2010

- C. Bertolli, D. Buono, G. Mencagli, and M. Vanneschi. An approach to mobile grid platforms for the development and support of complex ubiquitous applications. In *Handbook of Research on Mobility and Computing: Evolving Technologies and Ubiquitous Impacts*, 2011

- D. Buono, M. Danelutto, and S. Lametti. Map, reduce and mapreduce, the skeleton way. *Procedia Computer Science*, 1(1):2095 – 2103, 2010

- D. Buono, M. Danelutto, S. Lametti, and M. Torquati. Parallel patterns for general purpose many-core. In *Proceeding of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013

- D. Buono, T. D. Matteis, G. Mencagli, and M. Vanneschi. Optimizing message-passing on multicore architectures using hardware multi-threading. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 262–270, Feb 2014

- D. Buono, M. Danelutto, T. De Matteis, G. Mencagli, and M. Torquati. A lightweight run-time support for fast dense linear algebra on multi-core. In *Proceedings of the 12th IASTED International Conference on Parallel Distributed Computing and Networks*, 2014

- D. Buono, G. Mencagli, A. Pascucci, and M. Vanneschi. Performance analysis and structured parallelisation of the spacetime adaptive processing computational kernel on multi-core architectures. *International Journal of Parallel, Emergent and Distributed Systems*, 0(0):1–39, 0

- D. Buono and G. Mencagli. Run-time Mechanisms for Fine-Grained Parallelism on Network Processors: the TILEPro64 Experience. In *High Performance Computing and Simulation (HPCS), 2014 International Conference on*, 2014. To Appear

- D. Buono, T. De Matteis, and G. Mencagli. A high-throughput and low-latency parallelization of window-based stream joins on multicores. In *Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE 12th International Symposium on*, 2014. To Appear

Chapter 2

Background: hardware and software for parallel programming

In this chapter we describe the current state of the art concerning Chip Multiprocessors, both from a hardware and a software perspective.

We briefly classify and analyze the most common parallel programming tools available today, including interesting research work that hopefully gives an idea of ongoing research and future tools. We consider parallel programming in general, not only targeted to multi-cores, to give a summary of the overall current state of parallel programming.

Given the target of our thesis, we also introduce the most common parallel performance models, and the hardware-oriented models that inspired the work of this thesis.

Indeed, for the hardware perspective, we are not particularly interested in the current state, because of its fast evolution, but in some way we want to "forecast" how Chip Multiprocessors will be in the next years, to provide the basis for support and cost models that will be suitable for the next years. We therefore do not focus on a particular architecture, but present common choices for the various aspects of a CMP, and use current and future architectures as examples.

2.1 Chip MultiProcessor architectures

Chip Multiprocessors inherit their ideas from Shared Memory Multiprocessors. In fact, the first commercial multi-cores were just multiple multiprocessor-aware CPUs inside a single chip, connected with the same mechanisms used on multiprocessor configurations of that CPU, as with the Intel Pentium D processor in Figure 2.1.

Current CMPs are still the direct evolution of low-end SMP architectures of the late '90. These architectures were composed of 2 to 8 processors, and therefore used simple solutions that required small changes to uniprocessor CPUs: the system bus become shared among the processors, requiring only the presence of

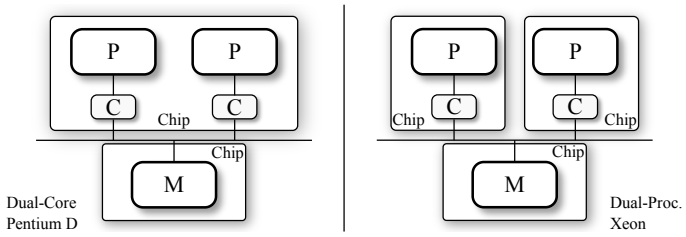


Figure 2.1: The first commercial Intel multi-core architecture: the Pentium D processor, compared with the corresponding single-core dual-processor architecture.

atomic operations and a snoopy-based cache coherence protocol to make these CPUs multiprocessor-aware.

Things evolved quickly, and engineers faced again the same problems found in large scale shared multiprocessors architectures: as the number of cores increase, the interconnection infrastructure become more complex; snoopy-based cache coherence protocols are not suitable, memory bandwidth is more important and so on.

In this section we will analyze the common choices for the important aspects of a multiprocessor (and therefore also of a chip multiprocessor), both for the current state and for a very likely future; throughout the section we present some architectures, incrementally as the choices are described, ending with a complete description of some current and future CMPs. In the multitude of processors, we chose three architectures from different domains, to give an overall idea of the evolution in various environments.

- **AMD Opteron 6100**[62]: this processor represent low and middle-end server domains. Each processor is composed of 12 cores, and can be used in 4 processor configurations. This is not a pure CMP, because it is made by two 6-core chips connected together as multiprocessor. There is therefore a mixture of CMP and MP inside the processor. It is represented in Figure 2.2.

- **Tilera Tile*Pro*64**[162]: specifically made for multimedia and network processing, this CMP is made of 64 general purpose cores, but cannot be used in multiprocessor configurations. It is represented in Figure 2.3.

- **IBM PowerEN**[83]: targeted mainly at network processing, it is a CMP made of 16 general purpose cores and some hardware accelerators; this architecture will be available in the following years in up to 4 processor configurations. It is represented in Figure 2.4.

These architectures are also compared with a large scale multiprocessor system: the SGI Altix 3000[184] (Figure 2.5), a NUMA built with Intel Itanium 2[124] processors, to give an idea of the differences between CMPs and MPs.

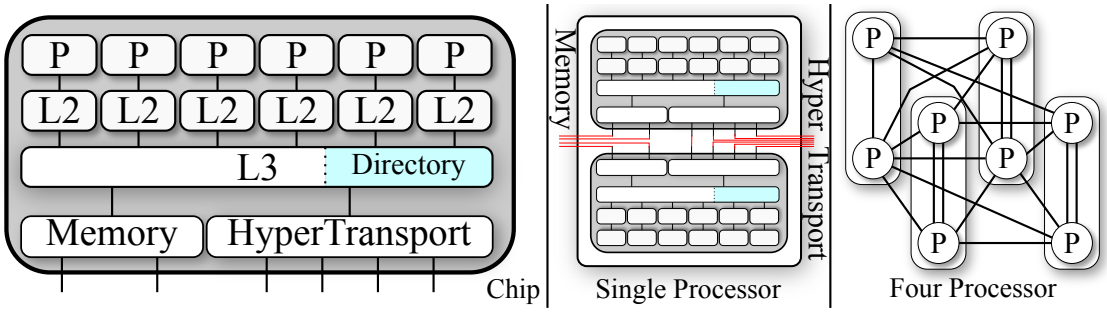


Figure 2.2: The AMD Opteron 6100 CMP and its 4 processor configuration.

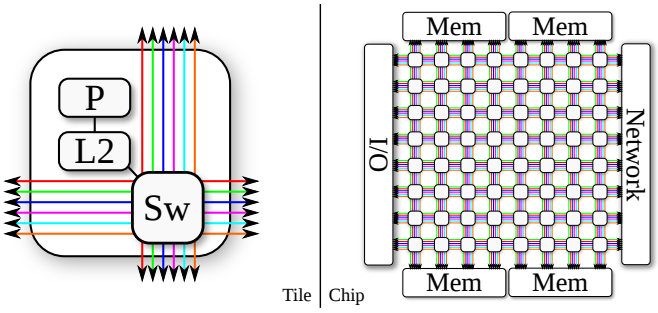


Figure 2.3: Single Cores of Tilera Tile*Pro*64 and the overall CMP.

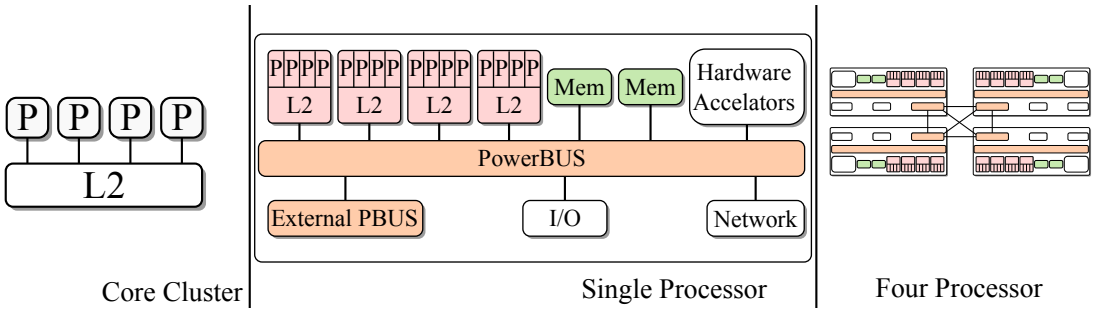


Figure 2.4: The IBM PowerEN: a core cluster, single CMP and 4-way configurations.

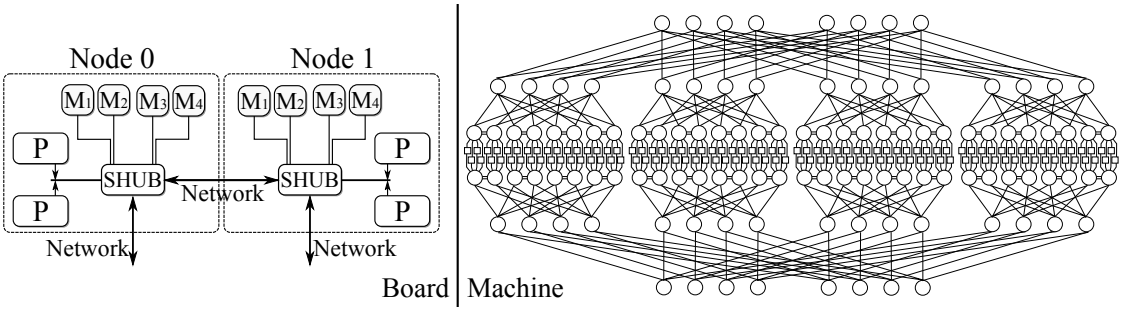


Figure 2.5: A large scale multiprocessor composed of single-core processors: the internal node architecture and its interconnection.

Special-Purpose Accelerators

The last years saw the strong emergence of accelerators explicitly developed to increase the performance of specific classes of applications. In fact, the accelerator concept has been always used in computer architectures to interface the general purpose CPU with specific interfaces (such as network devices, audio/video interfaces, and so on); however, up to some years ago these devices were devoted to the acceleration of a single, well specified, task. With the emergence of GPUs, these accelerators became more and more "programmable", up to the point that they are currently programmed *like* general purpose CPUs. Given the availability of a high number of (specialized) computing units, they are sometimes considered a specific class of many-core architectures. After the emergence of GPUs, other classes of accelerators (e.g. Digital Signal Processors - DSP or Field Programmable Gate Arrays - FPGA) adopted the very same programming approach, unifying the programming methodology for accelerators with the OpenCL parallel language.

Nevertheless, these *are not* general purpose parallel architectures, and as such they do not allow the indiscriminate porting of *any* application (at least with remarkable performance results), and they do not allow general interaction pattern among control flows (usually threads), so only specific parallel patterns can be instantiated. For these reasons we will not consider, in this thesis, this kind of special-purpose parallel architectures.

Characterization of a Shared Memory Multiprocessor Architecture

Shared Memory multiprocessors are parallel computers composed of a (possibly) large number of processors, usually identical, that share the main memory, i.e. each processor is able to address every part of the main memory. The most important characteristics of these architectures are:

- **Processor architecture**, that affects the performance of the sequential code;
- **Interconnection network**, among shared memory and processors;
- **Memory bandwidth and organization**, that affect the latency of load and store operations;
- **Atomic operations and synchronizations**, necessary to handle simultaneous accesses to the same memory address among processors;
- **Cache coherence**, required to keep private caches updated;
- **Number of processors** that, in the case of CMP is usually a constant value, while in MP represent the maximum number supported.

2.1.1 Processor architecture

Core complexity All the processors are 64-bit, pipelined and superscalar. Tile*Pro*64 and PowerEN have very simple cores with in-order execution, 2 (PowerEN) or 3 (Tile*Pro*64) execution units and no floating point units. Opteron and Itanium are much more complex, with floating point and SIMD units; Opteron is an out-of-order processor, three-way superscalar with 6 execution units. Itanium is surely the most advanced of the group, with its 11 execution units; it is, however, an in-order processor because of its VLIW instruction set that allow exploiting ILP without the high space and power consumption required by supporting out of order execution.

The different complexity is probably related to the domain of each processor: in general purpose server domains processors have to maintain good performance on sequential code and therefore inherit the complexity of high performance uniprocessor; when this is not necessary, simpler cores allow more cores per chip and lower power consumption. For this reasons we think that both kind of processors will remain in the future.

Caches A very important aspect of every processor is its cache hierarchy. All the processors have separated L1 for data and instruction; Itanium have separated L2 data and instruction caches, while in the others they are unified. Opteron and Itanium provides also an L3 cache. Cache sizes are different for each processor, with Opteron and Itanium having larger caches. Here CMP architectures start to diverge from classic uniprocessors: Opteron and PowerEN share a cache level among cores. For the PowerEN, the shared level is L2, while for Opteron is L3. In PowerEN the cache is not shared among *all* cores, but in small groups: there are 4 groups of 4 cores that share the L2 cache. This can be justified by the higher number of processors: sharing a L2 cache among 16 processors would make the cache slower, and therefore strongly limit the performance of the CMP. The Opteron, having only 6 cores and an L3, can share it among all the cores without performance problems.

Here again, we can highlight a trend for future architectures: as the number of cores increase, caches will not be shared among all the cores of the CMP, but eventually on small groups of cores (up to one as in the Tile*Pro*64).

Instruction Level Parallelism Another important aspect is how ILP (Instruction Level Parallelism) is extracted. Of the 4 presented processors, each one use a different mix of techniques to exploit its superscalar architecture.

The Tilera Tile*Pro*64 core is the simplest one, because it just uses a VLIW instruction set to let ILP being extracted by compilers. The PowerEN cores are RISC-based and in-order: they have no way to extract ILP from the code, and therefore use hardware multithreading, obtaining multiple independent instructions from different threads. The implemented multithreading is a 4-way SMT where instructions are fetched from each of the 4 threads at every clock cycle. Itanium is a VLIW processor like the Tile*Pro*64, but far more complex, and latest versions

implements a 2-way hardware multithreading that is, however, different from SMT because at each clock cycle instruction are executed only from the selected thread. Opteron cores implement out-of-order execution; the technique seems enough for the architecture, and therefore no hardware multithreading is implemented.

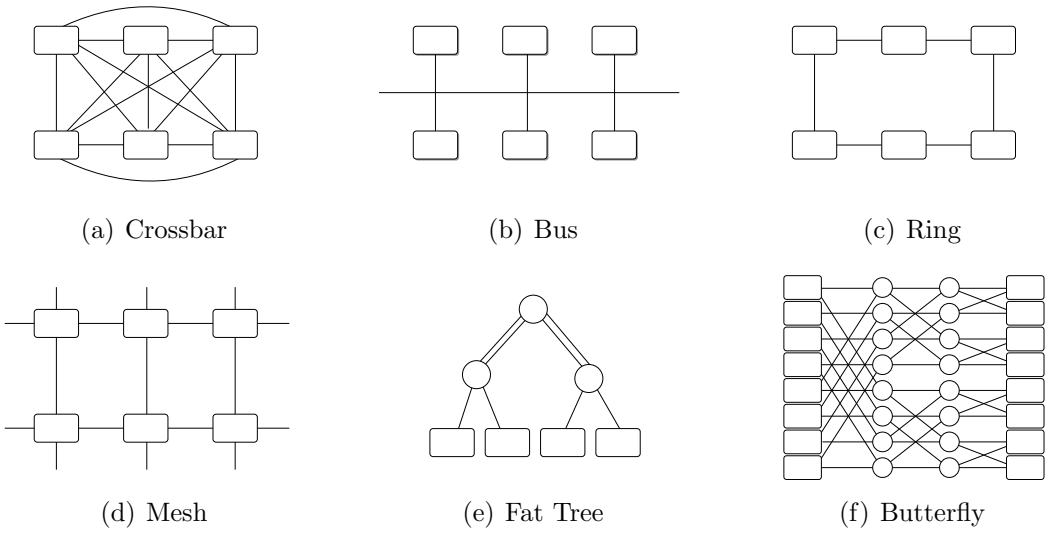
For a future trend, hardware multithreading (in the form of SMT) is having more and more success. In fact, except Opteron, all the others high end processors (SPARC, Power, Xeon) are using it, together with out-of-order execution.

2.1.2 Interconnection network

One of the most important part of a multiprocessor computer is the interconnection network that connects the processors and the shared memory. Processors exchange data each other, and the way (and the speed) this communication happens is mostly defined by the interconnection network.

The first two important interconnection networks are the crossbar and the bus. The first (Figure 2.6(a)) is an all-to-all connection, that therefore keeps the latency fixed, while the second (Figure 2.6(b)) represent a single link to which each node is connected. Here the latency is proportional to the link length (and therefore the number of nodes). Fat Tree (Figure 2.6(e)) is the most used interconnection network for large scale multiprocessor. The SGI Altix in Figure 2.5 uses two Fat Tree. This interconnection is basically a Tree that increase link bandwidth from the leafs to the root. This minimizes conflicts in the networks and gives a very high bisection bandwidth, maintaining a low degree for the nodes and increasing latency with a logarithmic scale. However, this is an interconnection for a group of N peers, i.e. NUMA architectures. To connect processors to a globally shared memory (i.e. UMA-SMP architectures) an interconnection that maintain these properties is the butterfly (Figure 2.6(f)) (that can be also used as a Fat Tree to allow inter-processor communications). Another common interconnection is the k-ary n-cube, especially in the form of rings ($n = 1$, Figure 2.6(c)), meshes (n=2, Figure 2.6(d)) and cubes (n=3). In these networks every node is both a computing and a routing node. Crossbar is obviously one of the most efficient networks, keeping latency constant and minimizing conflicts; however it can be applied to a limited number of nodes because of the number of links (n^2). Bus and rings are efficient and simple for a low number of nodes, as their latency is proportional to the number of nodes. The other interconnections become interesting when the number of nodes grows.

For the studied processors, the Tile*Pro*64, having many cores, uses a mesh that allow a good scalability and is easy to implement on chip. Inside the chip, the AMD Opteron have a crossbar among L2 caches and L3, while memory is directly connected to the shared L3; in multiprocessor configurations a partial crossbar (some links are missing) is used among processors. The PowerEN is in some way the most complex infrastructure: groups of 4 cores are connected to the shared L2 cache using a crossbar, and these groups are connected each other and to the memory using an enhanced bus. Multiple processors are then connected using a crossbar.



(a) Crossbar (b) Bus (c) Ring

(d) Mesh (e) Fat Tree (f) Butterfly

Figure 2.6: Common interconnection network.

It should be therefore clear that CMPs currently have a hierarchy of interconnection networks. While this trend could continue, we strongly believe that, as the number of core will increase, low latency interconnections like mesh, fat tree and butterfly will be also implemented on-chip[148], following the example of Tile*Pro*64 processor; in this case hierarchical interconnection will be probably removed.

2.1.3 Memory bandwidth and organization

One of the challenges with CMPs is keeping a good memory bandwidth and low latency. In common multiprocessors, each processor have its own interface to memory. Keeping this rule in CMPs is impossible because of pin-count problems: memory interfaces are (and will remain) proportional to the number of chips, and therefore considerably less than the number of processors. Opteron uses a single memory interface, shared among the 6 cores, PowerEN have two interfaces and the Tile*Pro*64 four. This can create memory bandwidth problems, as this must be divided among cores. Caches can help in keeping memory requests low, but are not always sufficient. If the CMP have a low number of cores or a single memory interface, we usually have an UMA architecture. This is the case of the Opteron (a single interface directly connect to the shared L3) and of the PowerEN (multiple interfaces but connected using a bus). With complex interconnections and multiple interfaces the architectures usually become NUMA, for example with the Tile*Pro*64, where the interconnection is a mesh and the memory controllers are placed at the borders of the chip.

Opteron and PowerEN, however, can be composed in Multi-chip architectures;

in this case we will have a hierarchical organization: SMP inside the chip, NUMA outside.

2.1.4 Atomic operations and synchronizations

Having a shared memory, synchronizations among processes (and processors) can be implemented accessing memory with atomic operations. However in CMPs synchronization using memory accesses could be costly, considering how "near" the cores are and how far the memory is. An inter-core synchronization (for example using the shared caches) could be much faster and allow finer grain parallelism. Moreover, with hardware multithreading, a way to synchronize threads running on the same core could offer even better results.

Many work in literature emerged in this area[71, 89, 167, 180], however in current CMPs the common way to synchronize is usually via atomic memory operations. PowerEN introduce a sort of "hardware" passive wait that stop executing a thread until a specified memory location is written by other threads. This can be considered a first step towards optimized synchronization mechanisms for CMPs. Tile*Pro*64 allows the program to directly use the mesh to exchange data (without accessing memory); this feature, while not technically a synchronization mechanism, can be used to implement fast inter-core synchronizations.

2.1.5 Cache coherence

An important problem in shared memory multiprocessors is keeping caches updated; this is usually solved implementing some automatic coherence mechanism that keeps caches updated. However this mechanism have a cost, which increase with the number of cores.

Caches maintain some additional bits for every cached block, which allows them to decide which operation perform in case of read/write. This state, however, must be updated when any processor perform operations on the same memory location; cache coherence mechanisms are usually divided in *snoopy-based* and *directory-based*, based on how caches are notified[64]. Snoopy-based cache coherence concept is very simple: memory access operations are notified to each processor (core) in the system. This way caches have no problems in updating the state of their blocks. However this requires broadcasting every access operation to anyone; it is convenient only with a limited number of cores, and is almost cost-free with bus interconnections (operations are always seen by all the nodes connected to the bus). For interconnections that are more complex a directory-based mechanism is usually chosen. The idea is to store somewhere information about which cache contain a block. This way, memory operations can be sent only to the set of caches that are actually interested. This approach minimizes traffic in the interconnection networks, however keeping and accessing the directory have a cost in term of space (the directory must reside in a fast memory) and increased memory access latencies.

Of the architectures studied, the Altix uses a directory-based cache coherence protocol; Opteron cache coherence is snoopy-based inside the chip (between L2 and L3) and directory-based among different chips. Here a configurable portion of L3 cache is used to implement the directory (Figure 2.2). In the PowerEN architecture the enhanced Bus provide cache coherence, supposedly via a snoopy-based mechanism, while the Tile*Pro*64 cache coherence is directory-based.

There are, however, some classes of computation that does not necessarily require an automatic cache coherence protocol, and permit efficient software-based cache coherence[1]. For these computations it is advisable to turn off automatic mechanisms and leave the work to the programmer; unfortunately this is usually not possible with current architectures. The PowerEN seems to allow the definition of non cache coherent memory pages; however, given the snoopy-based nature of cache coherence, the improvements should be low. Tile*Pro*64 allow defining cache-coherent domains, i.e. the set of memory locations that need automatic cache coherence. This way we could be able to remove any kind of cache coherence traffic and overhead.

2.1.6 Number of cores

Probably the most important property of a CMP, it is however strongly influenced by all the others presented before. Chip size is fixed, and a balance of all the components is important. This is why we still do not have an SGI Altix 3000 on chip: chip space is simply not enough to permit this configuration. The lower complexity of PowerEN and Tile*Pro*64 cores, for example, allows a larger number of cores inside a single chip (64 against 6 of Opteron). This imply a trend towards CMP with many simple cores, and it seems clear that in the future there will be two "families" of CMPs: complex cores with a relatively low parallelism, and simpler cores with higher parallelism. Complexity is probably the reason because Fat Tree and butterfly interconnections are still missing: meshes are simpler, take less space and for the current number of cores give performance similar to the other interconnections. However, in some years, a single chip will surely allow hundreds of complex cores, and then even more complex interconnection will have to be used.

2.2 Parallel programming on Chip MultiProcessors

CMPs can be programmed using all the tools previously available for shared memory multiprocessors. Considering that many of these are also given in "multiprocessor" configurations, with this solution we can expect good performance, aligned with that obtainable with comparable SMP multiprocessors. However, the specific features of CMPs (i.e. shared caches, hierarchical interconnections and fine-grained synchronizations) will not be exploited using these tools. For this reason a lot of

new parallel programming environments emerged; however, while specifically targeted at CMPs, many of these are very similar to already existing tools, and still fail to exploit specific features of multi-cores.

High level parallel programming is, in some forms, well known today, and even many of the tools we will describe here feature some "high level" concept. However, this is usually introduced with a "software-engineering" purpose, to speed up writing of parallel code. All the concepts related to performance portability and predictability, typical of structured parallel programming[18, 19, 139, 172] are completely missing. If a programmer want have an *efficient* code for a specific CMP, he have to write the program using low-level languages, and write code strongly related to the specific platform. In the same way, detailed performance predictions are not achievable, even with low-level programming, because detailed cost models of the architectures are still unavailable.

In short, a High Level Parallel Programming targeted at CMP is still missing, and programmers are thus forced to write "low-level" parallel code, strictly related to a specific CMP architecture, to exploit these features.

Parallel programming tools are usually divided in two large classes: *programming languages* and *libraries*. In the following sections we present the most important tools available for shared memory multiprocessing and those specific to CMPs.

2.2.1 Programming Languages

Parallel programming languages are usually extensions to well known sequential languages like *C*, *Java* or *Fortran*, that add specific constructs to define and coordinate multiple execution flows. However, some of them are completely new programming languages designed to write parallel programs.

In all of the presented, with the exception of *Erlang* and *Go*, cooperation among execution flows (threads, processes, etc.) is expressed using shared memory, and the mapping of flows on processing resources is done by the run time support of the language. While these languages performs well on SMP architectures, NUMA and distributed memory are difficultly exploited this way. For this reason some of them introduce the concept of a "Partitioned Global Address Space": the shared address space is logically partitioned, and for each flow a concept of "affinity" to a partition is given. With PGAS an execution flow conceptually have a fast local memory and a remote, slower memory, allowing good performance on NUMA and, with some limitations, on distributed memory architectures.

Most of these languages are to be considered "low level parallel programming languages", because programmers define all the execution flows and the cooperation among them. However, with respect to classical shared memory/message passing libraries, they are at a slightly higher level, because the programmer is somehow helped in defining the parallel program. The only notable exception is probably SMPSs, which is definitely at a higher level, but not sufficiently abstract to allow performance portability.

OpenMP[66] is one of the most common parallel programming language, and considered by many the de-facto standard for shared memory parallel programming. It is defined by a group of hardware and software vendors, including *AMD*, *Cray*, *HP*, *IBM*, *Intel*, *Microsoft* and many others.

Indeed it is not a complete programming language, but an extension that can be supported by *C*, *C++* and *Fortran* compilers. It defines an "accelerator-style" programming, where the main program is run sequentially and, in specific points, code is accelerated running in parallel.

The support to the most used sequential languages, the possibility to incrementally parallelize code and the fact that many compilers implement it, makes this "language" one of the most portable among platforms and architectures. However this "code portability" comes at the price of unpredictable performances: each compiler can implement the language in different way, potentially giving very different performance result even for the same parallel architecture.

Given its shared memory environment, OpenMP is specifically targeted to shared memory multiprocessors; some implementations support also distributed systems, but with very poor results. With CMP architectures OpenMP perform as good as in SMP architectures, but does not exploit any specific feature of CMPs.

Java [135] is a sequential programming language originally developed by Sun (now Oracle); it provides multithreading and RPC support, that can be used to write parallel applications for both shared memory and distributed memory architectures. However, while being an high level sequential language, parallel support is given in a very low-level fashion, even lower than OpenMP; moreover its sequential performance are usually lower than languages like C or Fortran and is therefore rarely used in high performance parallel programming.

Cilk [85] is conceptually similar to OpenMP; it is defined as an extension of the C language with annotations to express thread spawning and synchronizations. Thread scheduling is completely handled by the run time support of the language.

An interesting feature of this language is that, removing annotations, the resulting program *must* be a correct C program that sequentially execute the same computation of the parallel program.

The language itself is quite limited in expressiveness, but allows nested parallelizations. Produced applications can be executed only on shared memory architectures. Cilk is also one of the few academic parallel programming tools supported by the commercial industry: Intel is currently selling a specific version, called Cilk Plus, that specifically support its processors. Yet, to our knowledge, there is no specific support for CMPs in the run time, which is able to run on any shared memory architecture.

Berkeley Unified Parallel C [77] differs from the previous languages because uses a *Partitioned Global Address Space*. This feature allow it to be ported to the most important distributed memory architectures. Aside from that, the programming model is still a low-level shared memory environment, where execution flows synchronize with barriers and locks. Some collective operations are given by the language; for these operations the run-time support select the best among a set of defined implementations; the language offer, therefore, a (limited) approach to performance portability. Another interesting point of UPC is that its run time support is being optimized for CMPs[40, 134].

Erlang [177] is a message-passing parallel programming language originally developed by Ericsson. It is therefore very different respecting to the previous ones. In Erlang the set of execution flows operate in a local environment, and cooperate by using send and receive primitives. It is therefore mainly targeted at distributed systems, but recently shared memory implementations has been produced with results aligned to common shared memory programming languages, and very interesting results even on systems with a large number of cores such as the Tile*Pro*64 [190]. Giving a message-passing environment, this is still a low-level parallel programming language, exactly like the previously presented.

Go Programming Language [149, 86] is a recently introduced object-oriented language that exhibits a C-like syntax and greatly focuses on concurrency. Parallelism is automatically achieved by using "goroutines", functions specifically marked to be executed concurrently. Goroutines can exchange data by using asynchronous channels; thus, **go** highly resemble a low-level message-passing parallel language.

SMP Superscalar[141] and OmpSs[76] are interesting parallel programming environments based on compile time annotations (similar to OpenMP). OmpSs represent a sort of umbrella to capture the same concept in different architectures (SMPSs for multi-cores, GPUSs[17] for graphic processing units, ClusterSs[158] for clusters), with a uniform programming model.

Parallelism in SMPSs is obtained following a different approach, in which the programmer writes a program composed of tasks and dependencies among tasks. At execution, a data-flow graph of tasks is created according to their dependencies, and a run-time support is used to dynamically schedule independent tasks onto a parallel architecture. This approach is interesting because the programmer is only required to define tasks (i.e. functions) and dependencies among tasks; parallelism is automatically achieved by the run-time support. The same approach have been exploited in the past even by skeleton libraries (in particular Muskel[8]) and, during this thesis, by our research group on FastFlow[41] . Problems of this approach are mainly the selection of task sizes (in terms of computation time): theoretically, the smaller the tasks are, the higher the parallelism is; however small tasks requires

a larger dependency graph, and a higher impact of run-time overhead, that negatively affects the overall performance. The correct sizing usually depend on the target architecture, a factor that greatly influence the performance portability of this approach.

OpenCL [105, 155] targets specific architectures, and as such represent a conceptually different approach w.r.t the other languages. Initially defined to support GPUs, it now targets a wide range of accelerators, such as GPUs, FPGAs[65], DSPs[115] and classic multi-core architectures. Parallelism in OpenCL is a first citizen, as it is required to exploit these accelerators; however, the parallel model is quite different, as it does not allow indiscriminate cooperation among parallel entities. The main idea is that each parallel entity (called kernel) is run on the accelerator *independently*, while the host processor is in charge of distributing data to the accelerator. Of course this parallelism model (that strongly resemble a data-parallel map paradigm) is driven by the architecture of the original target accelerators (GPUs). In the following years the language has been extended to fully exploit new GPU characteristics that adds flexibility to the model, such as device partitioning, built-in kernels, atomic data and nested parallelism. The language represent, in fact an extension to C/C++; still, we do not consider this an high level parallel language, mainly for the absence of real parallel patterns and the need to explicitly decide which part of the code should be run, and how, on the group of accelerators available.

2.2.2 Libraries

With respect to programming languages, libraries are more heterogeneous, ranging from low level to high level parallel programming. However parallel libraries are somewhat limited with respect to programming languages, because static analysis and code optimization cannot be made inside a library. At the same time writing a library is very fast, compared to the work needed to create an optimizing compiler for parallel code.

There are many different libraries, but many of them share the same properties. Here we briefly present a selection that capture the various programming levels offered, to give an idea of the extended possibilities given by libraries, and at the same time, their limitations.

Posix Threads [50] is one of the most famous low level parallel programming library for shared memory environments. Can be found in almost every operating system, giving access to OS level threads and synchronization mechanisms. To our knowledge, there are no distributed memory implementations of this library. Moreover the library does not distinguish between CMP and MP.

Message Passing Interface [153] is the counterpart of the previous library, which allow processes with local environment to exchange data via send and receive primitives. Mainly targeted at distributed architectures, offer specific implementations for almost any high-performance interconnection network. At the same time, shared memory implementations are provided, that allows the use of MPI even on NUMA and SMP multiprocessors.

Intel Threading/Array Building Blocks [142] are two libraries provided freely by Intel to be used with their multiprocessors. They both define a set of parallel patterns to speed up parallel programming. Threading Building Blocks is mainly a stream-parallel library, while Array Building Blocks is targeted at data-parallel programming. However the given patterns are not very powerful, and it seems that there are no specific optimizations for CMP architectures.

FastFlow [12] is one of the first product specifically targeted at CMPs. It is a library created by our research group to exploit fast communications among cores of a CMP architecture. The library works at two different levels: it provides an efficient lock-free and wait-free communication channel that can be used directly by the programmer and, on top of this, a generic master-worker skeleton that allows the definition of stream parallel and data-flow computations. At the beginning of this thesis, FastFlow was not a complete high level parallel programming library, because it did not offer a wide set of patterns (just pipeline and farm). In the following years, with the introduction of the map data-parallel pattern and of a generic data-flow scheduler[41], and even the support for distributed systems[5], the library has become more complete. However, no cost models are used, and performance tuning is still a concern of the programmer.

Skandium [114] is a Java library to write shared memory parallel programs. It is a skeleton-based high level library, which provides a wide and almost complete set of nestable task and data parallel skeletons. While the author specifically target the library for multi-core architectures, the library is based on pure Java threads, making no distinction between MP and CMP; moreover the use of a Java language can partially limit its performance.

SkelCL [154] is a skeleton library targeting OpenCL. It allow the declaration of skeleton-based applications hiding all the low-level details of OpenCL. The set of skeletons is limited to data-parallel patterns: *map*, *zip*, *reduce* and *scan*, but it is unclear whether skeleton nesting is allowed. All the problems related to cost model are still present, and the library is limited (for our ideas) because of the target language and architectures.

2.2.3 Our vision of parallel programming

After this brief introduction to parallel programming, we end the section describing our vision on parallel programming environments, which will be extended in the next chapters. This idea finds its roots on the ASSIST[172] programming language developed by our research group some years ago. In our idea, low level libraries are very important, as they give complete control of the parallel application. This surely makes programming hard, but it is the only way to fully exploit a parallel architecture. They can be seen like the assembler language for sequential programming. Their existence is necessary to write good application; however, exactly like assembler, low level parallel libraries should not be used directly by programmers, but by means of compilers. In the opposite way, high level parallel programming libraries are somewhat less useful: while they offer a way to express parallel patterns, they usually cannot add platform-specific optimizations, analyze the user code or choose among a set of different implementations for the specific pattern.

We strongly advocate the need of a parallel compiler that, in addition to allow programmers to write high level parallel applications, analyze the parallel program to insert specific optimization (exactly like sequential compilers do), select the most effective implementation and provide platform-specific optimization. Many of them will be made using a cost model, to allow the compiler to compare different solutions. To our knowledge, no current parallel programming environments follow this methodology. However, among the plethora of research-oriented programming languages, we can find interesting approaches that somehow resemble our concepts. We analyze one of the most interesting, PetaBricks, to highlight the differences between the two programming environments.

PetaBricks: a (conceptually) similar approach

PetaBricks[14] is a novel programming language currently developed at MIT. The driving idea behind PetaBricks is that the best algorithm to perform a task may depend on the target architecture. In current languages the algorithmic choice is assigned to the programmer, as compiler optimizations are only in charge (at most) of optimizing the selected code (algorithm) for the specified architecture. Of course, a program *may* contain several algorithms to achieve a single task, but the selection of the proper algorithm is still a duty of the programmer. Some interesting approaches to auto-tuning libraries (i.e. software libraries that automatically select the best algorithm for the target architecture at compile-time) have been presented in the past: notable examples are the ATLAS[181] library for linear algebra and FFTW[84] to compute Fast Fourier transforms. Yet in these cases all the code required to select the best algorithm (by using specific benchmarks on representative input) was produced by the programmer of the library. PetaBricks tries to automatize the effort, with a language that allow the definition of multiple algorithms to perform a task, and an optimizing compiler able to select the best configuration for

the target architecture. PetaBricks is also defined as an "implicitly parallel programming language", as the algorithms are defined in such way that parallelism can be extracted *automatically* by the compiler. From this point of view the generated parallel code follow the same approach of SMP Superscalar: the code is composed of tasks, and a dependency graph among tasks is computed at compile-time; tasks are then executed, in parallel, by a dynamic scheduler that assign independent tasks to different cores.

The approach of PetaBricks is indeed towards performance portability, and confirms that current tools are not sufficient to achieve this level of portability. Nevertheless, several differences emerge w.r.t our ideas. PetaBricks works at the *algorithmic* level, while our approach is more oriented at exploiting different parallel version of the same algorithm, therefore targeting different (but partially overlapping) concepts.

The key difference, however, reside in how optimizations (and therefore the selection of the best choice) are achieved in the two approaches. PetaBricks rely in an autotuning system based on a specialized evolutionary algorithm called INCREA[13]. The "genome" consist in the search space of the possible algorithms (and algorithm combinations); INCREA evaluate fitness by running candidate programs on representative inputs. Our approach in optimization is completely different, as we use cost models to evaluate each solution; therefore we do not need to run the (parallel) program on sample inputs, but to search, in the space of possible parallelizations, the one that minimizes the cost model. We believe that our approach is also more feasible to exploit *adaptivity* on dynamically changing environments, where the cost model can be used to *drive* reconfiguration by estimating the final result at run-time without the need of specific, possibly heavyweight, benchmarks. Models allow us to *predict* the behavior and let us use well known approaches such as the optimal control theory, introduced in a previous work of our research group[129].

2.3 Performance model for multiprocessors

Given the focus of the thesis, we also briefly present the most important models currently used to predict the performance of a parallel application, and explain why they are not suitable for our work. Then, we introduce the reader to hardware-oriented performance models and their use in the past years.

2.3.1 Algorithm oriented performance models for multiprocessors

The most common parallel performance models are built for parallel algorithm designers, which are not interested in particular architectures, but looks for algorithms that perform well in general. The models are therefore based on an asymptotic prediction of the performance, exactly as the complexity order is analyzed in sequential

algorithms.

The idea of allowing a simple and machine-independent study of the algorithm is indeed in contrast with our idea of a detailed and machine-dependent prediction. These models are not much useful for us and are presented just for the sake of completeness.

PRAM

The Parallel Random Access Machine[80] models a shared memory abstract machine with a possibly infinite number of processors. The parallel computation is described as a sequence of *Read - Compute - Write* executed by each processor. These sequences are synchronously executed by each processor, which can read and write any location of the shared memory. Memory accesses and synchronizations among processors are considered cost-free, and the time complexity is given by the number of sequences executed by the longest running processor. This model obviously represent a very abstract parallel machine, and can be useful only for asymptotic study of algorithms

BSP and Multi-BSP

The Bulk Synchronous Parallel[169] (BSP) is another abstract parallel machine. It is similar to PRAM, except for the fact that consider synchronization and communication costs. In BSP each processor is assumed to have a local memory, and cooperate with the other processors exchanging data. A computation is defined as a sequence of "superstep"; in each superstep a processor makes a local computation, send required data to other processors and then wait the end of the superstep of other processors. The cost of each superstep is defined as $max_{i=1}^{p}(w_i) + max_{i=1}^{p}(h_i g) + l$, with w_i is the cost of the local computation, h_i the number of messages sent, g the cost of sending a message and l the cost of the barrier. The complexity of the entire computation is simply given by the sum the supersteps. Just adding these few parameters made the model much more detailed and realistic. However, using synchronous superstep greatly simplify the model, and differ from multiprocessor architectures where processors are independent and can synchronize each other with point-to-point synchronizations.

With the emergence of multi-core architectures the author of BSP proposed an extension, called Multi-BSP[170], that address the existence of a memory hierarchy, and thus different levels of sharing among processors. Multi-BSP is a multi-level model, in which each level is composed of supersteps, retaining the initial ideas of BSP. According to the author itself, the model is still proposed to evaluate the complexity of parallel algorithms, and not to precisely estimate the performances of the algorithm on specific architectures[170]:

> The goal here is to identify a bridging model on which the community can agree, one which would influence the design of both software and

> hardware. It will always be possible to have performance models that reflect a particular architecture in greater detail than does any bridging model, but such models are not among our goals here.

LogP

LogP[63] is one of the last algorithm oriented performance models developed. Recognizing how far were previous models from real parallel machines, the authors tried to describe an abstract machine that, while being easy to model, could be similar to real hardware. Massively parallel machines are generally distributed memory multiprocessors; LogP model that kind of parallel architectures by defining the number of processors(p), the communication bandwidth(g), the communication delay(L), and the communication overhead(o). While considering the properties that roughly define a parallel machine, the model abstract from the internal hardware of the nodes and from the details of the interconnection network (routing, topology, etc.). Programs are then defined as asynchronously cooperating processes that can exchange data with a well defined cost. It is a sort of extension of BSP, where programs do not follow synchronous steps and the cost of exchanging data is more detailed. It is still, however, a rough model that can be useful to write good parallel algorithms, but cannot be used to compare architectures or different implementations of the same program.

2.3.2 Hardware-oriented performance cost models

The idea of specific performance models for parallel patterns was introduced a few years after the introduction of skeletons, and already exploited in P^3L[18] to select the so-called "implementation templates". These, however, modeled the performance of the parallel pattern with an approach similar to that of LogP, taking the sequential code as a "black box", with specific, immutable, characteristics. While this is usually true for distributed architectures, where each program is executed in a different machine that is loosely coupled with the others, in multi-core architectures the situation is really different, as we observe several kind of interactions and shared resources that may affect the performance of the sequential code, and thus affect the final result in term of performances.

We therefore extend the original concepts presented in P^3L[139] and employed in SkiE[19], ASSIST[172] and initial versions of ASSISTANT[32], by introducing an *architecture model*, that allow us to predict the performance degradation that occur when multiple processes are executed on the same multi-core chip. It is widely known that the major source of degradation is the sharing of the memory subsystem. One of the first to analyze this form of degradation has been Bhandarkar[36], that modeled the processor-memory subsystem with the queueing network in Figure 2.7, where each processor generate a memory request and then stop its execution waiting for the response.

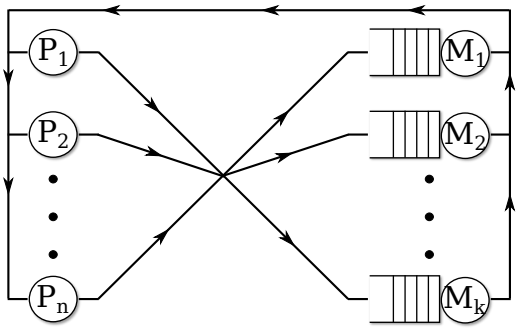


Figure 2.7: Bhandarkar's queueing network to model the memory contention.

From his seminal work many extensions were produced, mainly in the '80, to model the emerging parallel architectures of that days[3, 97, 122, 164]. The focus of these works was, however, usually different from our perspective: most of them are more interested in mathematically sound approximations, without caring of the possible applications of the model, or in analyzing the asymptotic behavior of an hypothetical architecture, by selecting the parameters of the system in a realistic, but not program-driven, way [3, 176, 188].

However, there were a small amount of work that proposed, as we are doing, to use these models in the performance evaluation of a parallel program [2]. Still, to our knowledge, none of these addresses the problem *specifically* for multi-cores.

2.4 Summary

In this chapter we introduced the reader to the world of chip multiprocessors, and in particular to their hardware features and the programming models currently used. Given the level of details of certain works, it was not really feasible to introduce and describe all of them here. We therefore decided, for the sake of readability, to keep this chapter at an introductory level, and further describe the important works in the following chapter, when the reader will have the tools to understand them.

We presented the current state-of-the-art in CMPs and the most feasible future trend, in which the amount of processors per chip will increase to a point that programming these chips will become even more difficult.

We highlighted the most important problem in software development, which is the absence of complete environments especially targeted at multi-core, able to fully exploit these architectures without a manual intervention of the programmer.

Finally, to control the complexity of these and future chips, we advocate the need of a different approach to performance modeling, which should be used not only to evaluate the asymptotic performance characteristics of an algorithm, but also to compare, at a very fine level, the performance of different implementations.

Chapter 3

Structured parallel programming to solve the multi-core problem

In this chapter we present our methodology to solve the problems of parallel programming on multi-cores.

We introduce the general concepts of a structured programming model and motivate the need for such kind of programming. Then, we describe our model, that finds its roots on the ASSIST programming language developed by our research group some years ago, and compare it with respect to other classic structured programming models. We will not provide, however, a precise definition of the language, as this is beyond the scope of this thesis: the purpose of this chapter is to understand the *level of abstraction* provided by the model. Abstraction is, indeed, the strength of our approach, that let us *automatically* and *transparently* create and transform parallel code without user interactions. In the last part of the chapter we will introduce the workflow of our parallel compiler, highlighting the open research points that will be partially addressed in this thesis.

3.1 The need for high level parallel programming

Since its introduction, parallel programming was strictly related to HPC environments, in which programmers were usually willing to write parallel code by mean of low level libraries that, giving a complete control over the parallel application, allowed them to manually optimize the code and exploit at best the parallel architecture. This programming methodology exposed its first problems with the emergence of cluster and grid computing, when the parallel architectures become dynamic and heterogeneous, therefore limiting the possibilities of ad-hoc optimizations. The recent massive introduction of parallel architectures in every computing device, and thus in every computing sector, critically exposed the lack of proper tools to easily implement a parallel application: the industry cannot afford the cost of re-writing (or even re-tuning) an application for every available computing architecture.

For this reason a lot of parallel programming tool were introduced in the last decade, both by the academic and by the industrial world. Despite the increasing effort (especially of the scientific environment), automatic parallelizers (i.e. compilers that automatically extract parallelism from a sequential program) are still ineffective in fully exploiting parallel architectures, rarely providing good speedups even on 4 and 8 cores [52, 82]. To the current state of the art, in short, a proper rewrite of the program is needed to exploit parallelism. In this case, a proper mix of *ease of use* and *performance* is still the main concern of researchers. It is widely acknowledged that one of the mayor problems to be addressed by a parallel model is *portability*. Indeed, portability for parallel programs exhibit a twofold nature:

1. **code portability** that is, as for sequential programs, the ability to compile and execute the same code on different architectures;

2. **performance portability** that represent the ability to efficiently exploit the underlying parallel architecture.

Performance portability is indeed very important: a parallel program that does not exploit the architecture must be rewritten, thus nullifying most of the benefits of code portability. It is widely acknowledged by the scientific literature [69, 171, 152] that performance portability is achievable only by using a high-level approach to parallel programming: exactly as in sequential program, were portability is guaranteed by sequential high-level languages, we need to define parallel constructs that allow a proper compiler to produce efficient code for any architecture.

In other terms, by using a high-level parallel model, we are able to describe our parallel application and be sure that it will perform reasonably well on the wide choice of parallel architectures available today.

In the previous chapter we described a wide set of programming models; beside from the common low-level libraries (i.e. Posix Threads and MPI), all the current tools express some characteristics of high-level parallel programming, i.e. help the programmer by easing the burden of writing parallel applications. For example, OpenMP uses a shared-memory programming model, but allow the programmer to extend sequential code by mean of annotations, without explicitly writing the parallel threads. The programmer must, however, know basic concept of the resulting parallelization to ensure program correctness. OpenMP also allows specific architecture-driven optimizations, i.e. the same annotated code can be "compiled" in different programs depending on the target parallel architecture. However these optimizations are usually quite simple; furthermore, in many cases a detailed knowledge of the parallel implementation and proper annotations or code reorganizations are required to ensure good parallel performance.

The same concept apply, more or less, to all the languages and libraries: they help the programmer in many ways, especially hiding details typical of low-level parallel programming, but they do not allow that kind of *performance portability* described before.

3.2 Structured parallel programming

Structured parallel programming is probably the most interesting class of high-level parallel models. It started with the concept of algorithmic skeletons defined by Cole [61] and has been successfully applied basically in any possibile parallel environment, starting from clusters[67] and shared memory machines[114], to grid[7], cloud and pervasive environments[32].

Two of the most important point of structured parallel programming are the ability to *automatically* create different parallel implementations starting from the high-level description, and the *parametric* nature of the produced code, that is able to run with different parallelism degrees. These points are the basic building blocks to ensure *performance portability* on the various architectures. Structured parallel programming also allows *composability*: a parallel code can be mixed with others, such that an application can be described as a collection of parallel kernels, instead of a single, big, large parallel code. Composability also allow reuse: the same kernel can be reused inside different programs with no modifications.

Our research group history in structured parallel programming is quite long, starting with the P^3L skeleton language in 1992, and culminating with ASSIST in the last years. These projects incubated many interesting developments for parallel programming: we implemented, among the others, parallel code restructuring[4, 6], to better exploit the composition of parallel kernels; efficient fault tolerance [35], and dynamic reconfigurations up to self-adaptive programs[127, 125], i.e. programs that are able to exploit performance portability *dynamically*, at run-time, to better fit dynamic environments such as grids or clouds. Finally, we also extended the concept of High Performance Computing to Grid computing[7] and lately to Pervasive Grids[32].

We never, however, really focused our efforts in multi-core and shared memory architectures in general. Our experiments with FastFlow[9] demonstrated the need, and the possibility, of multicore-specific optimizations.

3.2.1 Parallel Paradigms

A parallel paradigm (also called parallel pattern) is the core concept of structured parallel programming; it is, in short, a well-known pattern of interaction of a parallel code. We can identify a small number of paradigms that can be deeply studied for the target architectures, to provide an efficient implementation. One of key concept of paradigms is that a single pattern can be used to describe just a small part of a complex application; their strength lies in the fact that multiple paradigms can be easily mixed together to describe complex applications.

This is the basic idea behind structured parallel programming: expressing the parallel code as a composition of widely-studied "building blocks". This allows some sort of *separation of concerns*: the *application programmer* is in charge of defining its applications, while a *parallel programmer* defines the *parallel implementation* of

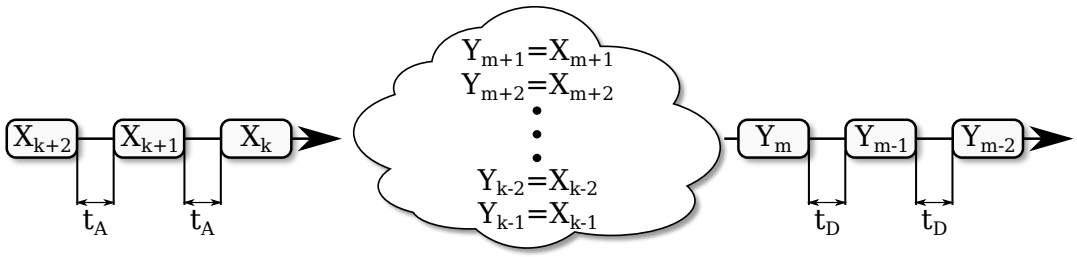


Figure 3.1: Graphical representation of a generic Stream-Parallel Pattern.

each building block (and eventually some notable compositions) on the supported architectures. The latter is done per-pattern, and not per-applications, thus incredibly relieving the amount of work for both the programmers.

Parallel paradigms can be divided in two conceptually different classes depending on the way parallelism is exploited inside the pattern: *Stream-parallel paradigms* and *Data-parallel paradigms*

3.2.2 Stream Parallelism

This class contains those patterns that exploit the presence of a *stream*. A stream is a (virtually infinite) sequence of input elements to be processed; parallelism is simply obtained by processing multiple elements *concurrently*. An important characteristic of streams is that the elements *arrive* with a certain time distribution (i.e. the sequence is not entirely available at the beginning of the computation, but is produced *during* the computation). This represents an important limiting factor of the parallelizations: whenever we run the parallel code, we cannot compute elements at a higher rate than the one they arrive. In other words, although the stream is virtually infinite, we have a limitation in the performance (and the possible number of processors used) of the resulting parallelization.

It is important to note that stream parallelism does not speed-up the computation of a single element, but the computation of the *stream*. In terms of performance evaluation this means that a stream-parallel pattern does improve the throughput (i.e. the amount of elements computed per time unit) but not the latency (i.e. the time needed to execute the computation on each element). When latency is important from a performance point of view, this kind of parallelism is ineffective.

The concept of a generic Stream-Parallel pattern is depicted in Figure 3.1, where we define the average arrival time as t_A, and the average departure time as t_D. In the optimal case, $t_D = t_A$, meaning that our stream-parallel pattern is able to fully support the incoming throughput with no performance degradation.

The existence of a large sequence of input elements is a necessary precondition in order to apply these parallelization techniques: no performance enhancements can be obtained if we consider a single or a limited set of input elements, because of the

limited possibility of computing them concurrently.

The need of a stream can appear as a strong limitation of this class of paradigms; still, there exists a lot of computing environments that works with streams, such as video processing (where there exist streams of frames), network filtering (streams of packets), signal processing (streams of signal samples) and so on.

The two most representative parallel paradigms are the so-called *task-farm* and *pipeline*. While the basic idea is the same (computing more input elements at the same time), they strongly differ on the way parallelism is obtained.

3.2.2.1 Task-Farm

The Task-Farm parallel pattern is probably the simplest, yet most used, paradigm, as it represent the *functional replication* of the sequential code as a whole.

The idea is pretty simple: let say our code behave as a function:

$$y = F(x)$$

we replicate the code of F on n different processors (usually called workers); then, when an input element is received, it is forwarded to one of them, by using some selection policy to guarantee that all the workers will take some element, as depicted in Figure 3.2.

It is simple, because it usually do not require a rewrite of the sequential code (that is just *encapsulated* in a farm pattern) and allow for virtually *infinite* parallelism (i.e. we can use as much processors as we want, respecting the input stream arrival time). The main limitation is that, to guarantee correctness, the code have to behave like a *pure function*: $F(x_i)$ cannot depend on $F(x_{i-1}), ..., F(x_1)$. To let this paradigm perform as expected we would also need to balance load among the function replications (i.e. ensure that all the processors will take approximately the same amount of work); furthermore at the end of the computation a correct element ordering is generally required, to guarantee that the resulting computation

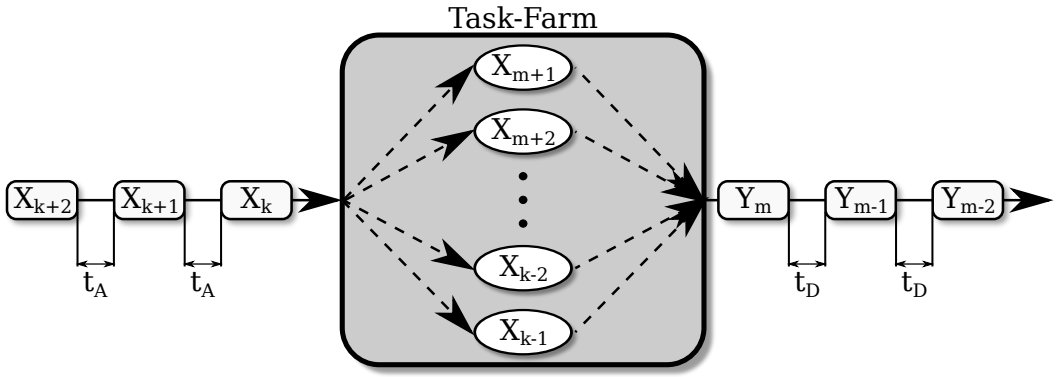


Figure 3.2: Graphical representation of a Farm pattern.

is equivalent to the sequential. These operations are usually executed by *support entities* that act as an interface between the farm and the outside world. When present, they are called *Emitter* and *Collector*.

3.2.2.2 Pipeline

In contrast to the previous paradigm, the pipeline does not represent a functional replication but a *functional partitioning*: the sequential code is *divided* in multiple pieces executed concurrently.

The application of the pipeline paradigm requires some knowledge of the form of the sequential computation, that is the sequential computation must be expressed (or rewritten) as the composition of n functions:

$$F(x) = F_n(F_{n-1}(...F_2(F_1(x))...))$$

In this case, a pipeline parallelization consist in a set of (at most) n entities, each executing one (or more) of the n functions. An entity (usually called "pipeline stage") will receive each input element and will compute its function on it. The output of each entity is sent to the next one, respecting the function ordering (i.e the output of F_1 is sent to F_2), so that the output of the last stage (i.e. F_n) correspond to $F(x)$, as depicted in Figure 3.3.

One of the most important benefits w.r.t the farm paradigm is that code of the Pipeline do not have to be *stateless*, i.e. F_i do not need to be a *pure function* and may depend on the previous computations (but only of F_i, not of other stages). This comes, however, at a cost: the programmer need to describe its sequential code as a composition of functions, and the number of functions limit the maximum parallelism of the application (i.e. if the resulting parallel pattern is a 3-stages pipeline, it will exploit *at most* three processors).

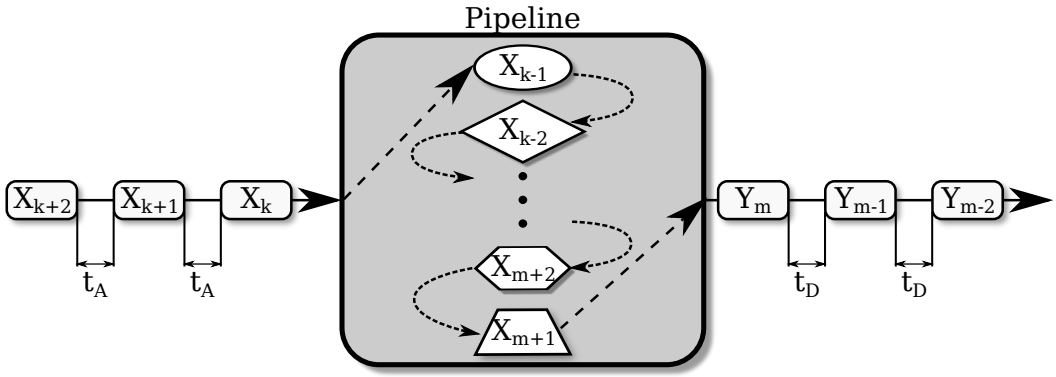


Figure 3.3: Graphical representation of a Pipeline Pattern.

3.2.3 Data Parallelism

In contrast with the previous, this class of paradigms extract parallelism *inside* a single computation, by partitioning the data structures and, by reflection, partitioning the computation. The most common partitioning approach is defined by applying the *owner computes rule* to determine, once the data structure is partitioned, which entity should perform the computation on each piece of data.

Consider, as an example, an application defined on a matrix, as in Figure 3.4. The matrix is partitioned, let us say in blocks, and each block is assigned to one of the n different processors (usually called workers). Following the owner computes rule, each worker is in charge of calculating its partition. Obviously, depending on the algorithm, this will require data from other partitions (in the example, the points inside function parameters), and therefore workers: we define these as *data dependencies*, and one of the most important duties of a pattern implementation is indeed handling data exchange among the workers, when needed.

Usually the number of partitions is not fixed a-priori: if more processors are available, the pattern can exploit a *finer* partitioning (smaller blocks in the example) to increase the number of partitions. This approach, however, is not always applicable: first of all, there normally exists a *minimum partitioning*, determined by the items that compose the data structures (single matrix elements in the example); moreover, the smaller is the partition, the smaller is the calculation time per worker, so that this approach is profitable (in terms of performance) only up to a point.

More formally, a data parallel computation is characterized by partitioning of data structures and function replication. It may happen, however, that while some data structures are partitioned, others are instead replicated among the workers to allow better performances (we will see some examples later).

Theoretically any algorithm can be parallelized by following a data-parallel approach; however, in practice, many data-parallel programs may suffer of bad performance scalability because of a high number of data dependencies or a low amount of inherent parallelism. Nevertheless, this class of patterns is very powerful.

If we follow the original *algorithmic skeleton* definition, dependencies among

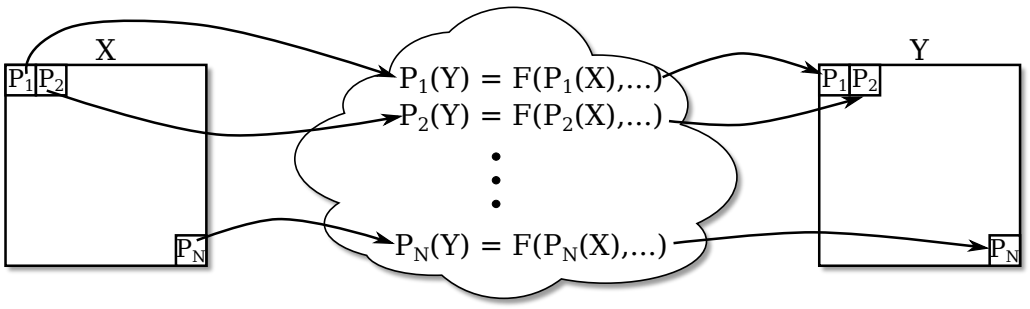


Figure 3.4: Graphical representation of a generic Data-Parallel Pattern.

workers determine the parallel pattern, so that any data dependency scheme will produce a new pattern. Fortunately, many algorithms (even from different computing sector) produce the same dependency scheme, so that we are able to define a small number of patterns that actually cover most of the algorithms.

It is important to notice that data-parallel paradigms *may* also works with streams, but differently from stream-parallel paradigms, they compute one element at a time. From a performance evaluation point of view, a data-parallel pattern improves the calculation time of a single execution; in the presence of a stream, this parallelization allows better latency and, by reflection, better bandwidth.

For the sake of completeness, we present the two most used patterns, *Map* and *Reduce*, and the general class of *Data Parallel with Stencil*, that basically contains all the other algorithms.

3.2.3.1 Map

This is the simplest case of a data-parallel program, in which the resulting workers remains independent (i.e. there are no data dependencies):

$$y_1 = F(x_1), y_2 = F(x_2), ..., y_k = F(x_n)$$

Where y_i and x_i represents the "minimum partition" of the data structure, and therefore the maximum amount of parallelism; Figure 3.5 graphically represent this pattern.

The applicability of this pattern usually depend on the algorithms; sometimes different partitioning of the data, or replication of some structures permit the use of this paradigm even in algorithms that, from a first analysis, do not resemble the pattern.

There are also notable examples that naturally fit this paradigm, such as many vector operations (*scalar-vector* multiplication, vector sum, etc), matrix multiplication (in which the result matrix is partitioned, the input matrices replicated), the Mandelbrot Set calculation, and many others.

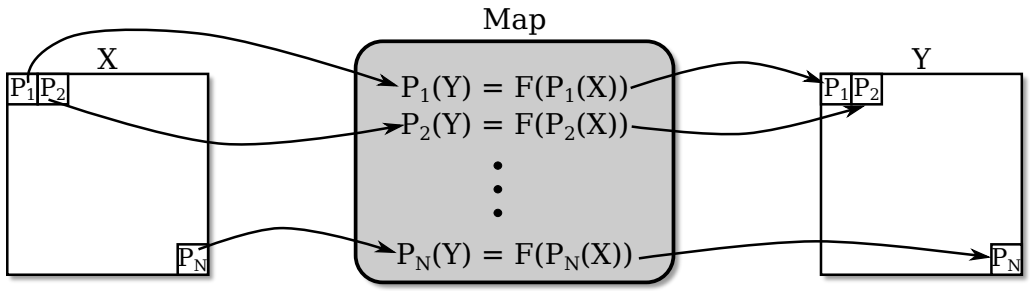


Figure 3.5: Graphical representation of a Map Pattern.

3.2.3.2 Reduce

A Reduce pattern is applicable every time we have a computation of the form:

$$y = x_1 \oplus x_2 \oplus ... \oplus x_k$$

where the result is a single value obtained by applying a function ($\oplus$) to all the elements of the input data structure. To ensure correctness $\oplus$ also have to satisfy the *associative* property.

This can be easily parallelizable (Figure 3.6) by partitioning the data structure in n workers (parametric but limited by k), each one performing a "local" reduce on their partition, followed by a "global" reduce of the results of each worker. The global reduce can be executed in several ways (even in parallel) by one (or more) of the workers.

Notable examples that naturally fit this paradigm are many vector- or matrix-based operations such as the maximum and minimum of a vector, the dot product and many others.

3.2.3.3 Map + Reduce, a notable composition

Many algorithms can be defined as a composition of two steps, in which at first a function is applied to all the elements, and then the results are merged by using some reduction function. This is one of the most important examples of that *composability* allowed by parallel patterns: we can straightforwardly represent this class of algorithms as the composition of a *Map* and a *Reduce* pattern. This originated the famous Google MapReduce[72] programming environment (and its open-source implementation Hadoop[1]), that saw an enormous success in the last years. Map+Reduce is an interesting example also from the performance point of view because, although being a composition of two patterns, there exists optimizations, and thus efficient implementations, that threat it as a whole (and not as two distinct pat-

[1] Available at `http://hadoop.apache.org/`

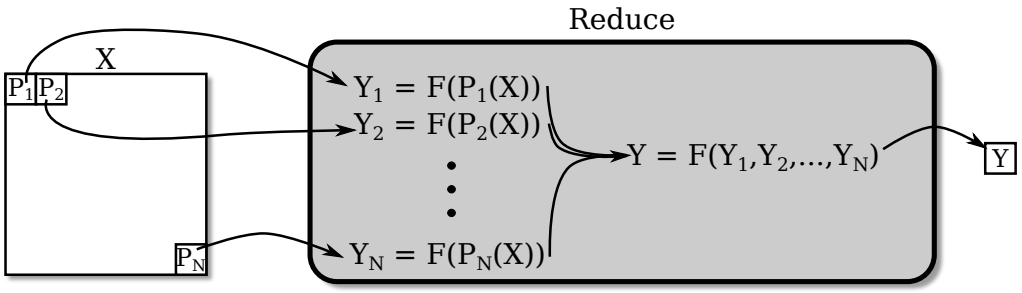


Figure 3.6: Graphical representation of a Reduce Pattern.

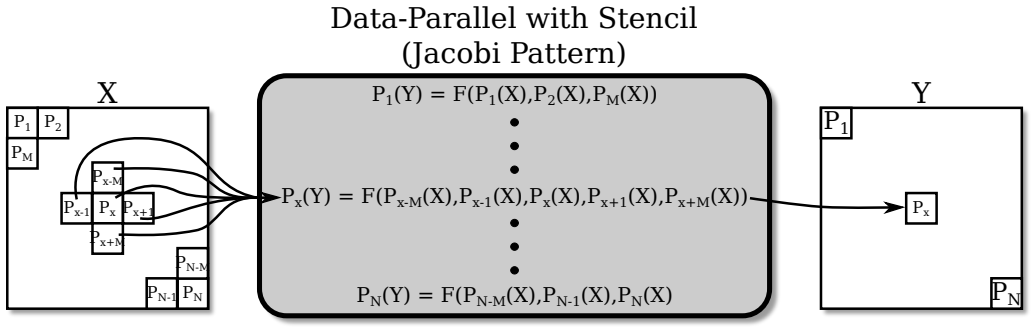


Figure 3.7: Graphical representation of the Jacobi Pattern, a data-parallel with a static fixed stencil.

terns): this is another aspect that a structured parallel programming environment should be able to exploit.

3.2.3.4 Data-Parallel with Stencil

This is not really a pattern, but a class of patterns, that contains all those that require data exchange among workers. The name derives from the data dependency shape that is called "stencil"; each stencil represent a different paradigm, some widely adopted, others limited to single algorithms.

A Stencil (i.e. the data dependency shape) can be

- **Static fixed**, if the shape is defined at compile-time and remain the same throughout the computation.
- **Static variable**, if the shape is defined at compile-time but changes during the computation (this is a common phenomenon in some iteration-based algorithms such as FFT).
- **Dynamic**, if the shape is defined at run-time, depending on data structure values.

Of the three, a *dynamic* stencil cannot be captured by a parallel paradigm, because the dependencies are not known until execution. The others are instead very common in skeleton-based environments: P^3L, for example, included "geometric" (static fixed) and "tree" (static variable).

A notable example of a "static fixed stencil" is the *Jacobi algorithm* (depicted in Figure 3.7, used for solving partial differential equations, where the *shape* is represented by the neighbor of each point. The Fast Fourier Transform, on the other hand, is probably the most common example of "static variable stencil", because the shape changes at each iteration of the algorithm, but following a well-known mathematical rule.

3.2.4 Stencil Transformations

An important yet still quite underestimated point in structured parallel programming is the possibility of transforming stencil. In fact the *high-level stencil*, i.e. the stencil originated by the "minimum partitioning", can sensibly differ with the stencil obtained *after* the partitioning in workers. In particular, there are two mayor operations that allows us to change the stencil (i.e. the shape of data dependencies).

Worker partitioning

An important aspect is that, starting with the "minimum partitioning", we usually have to increase partition sizes to fit the exact number of workers; this *can* change the stencil, even significantly. For the sake of simplicity we analyze this on the the *Jacobi Algorithm* introduced in the previous section. We previously described the 4-points stencil among each point and depicted a block-based partitioning that maintains the same stencil. However, in Figure 3.8 we show what happen if the point partitioning is done per-row or per-column: the resulting stencil (among partitions) is a 2-points stencil, thus reducing the number of dependencies. In general, this method can effectively produce a completely different stencil by means of a different worker partitioning, for many algorithms.

Data Replication

In many algorithms we can efficiently transform the application, completely *removing* the stencil to obtain a Map data-parallel, at the cost of a (usually small) replication of data. The basic idea is that data dependencies defined only on *read-only* or *input* data can be replicated at the beginning of the computation to make the workers independent. Notable examples are represented by most image filtering algorithms that, with a proper replication, can be described with the map pattern. Let us take, as example, the *Mean Filter*, a smoothing technique [182]. The basic idea of the algorithm is to smoothen an image by replacing the value of each

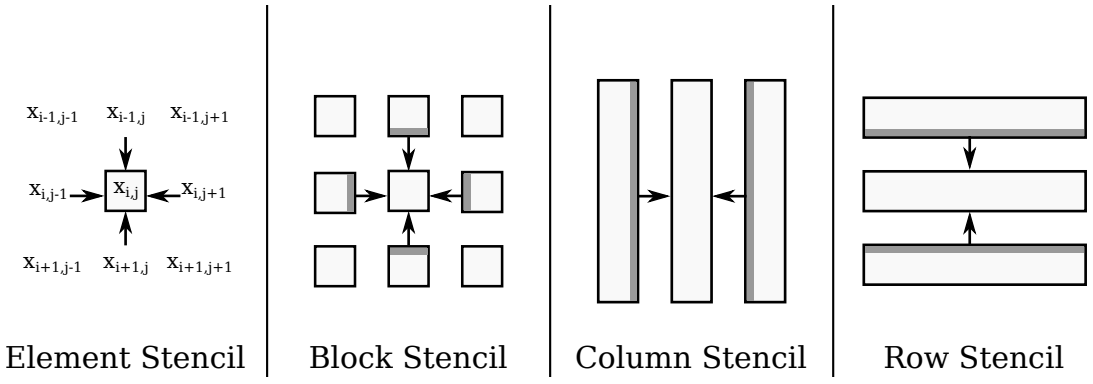


Figure 3.8: Partitioning of the Jacobi Pattern that produce different stencils.

pixel with the *average* of the group of points *near* the pixel itself. For example, the smoothing of pixel $x_{i,j}$ with a 3x3 group size is obtained by

$$y_{i,j} = \frac{x_{i-1,j-1} + x_{i-1,j} + x_{i-1,j+1} + x_{i,j-1} + x_{i,j} + x_{i,j+1} + x_{i+1,j-1} + x_{i+1,j} + x_{i+1,j+1}}{9}$$

If we consider the standard partitioning rule we mentioned earlier, we would have that each point on the border of a partition will depend on data from other partitions, implicating some kind of dependencies. In practice, however, we can solve this problem with replication if we define partitions that partially *overlap* each other. In the particular example we can notice that by enlarging the partition of 1 pixel in width and in height, without changing the pixel assigned by following the owner-compute-rule, each worker will have all the data needed to compute, making the algorithm a perfect match for the map paradigm. As depicted in Figure 3.9, we basically solve the data dependency problem by selectively replicating parts of the data structure.

The same concept can also be applied partially (i.e. when only some of the data structure are read-only), resulting in stencil transformations that, however, do not always produce a Map pattern.

Stencil Transformations and structured parallel programming

Stencil transformations can prove very effective in increasing the performance of a parallel application, and as such they should be considered *first citizen* of structured parallel programming. Nevertheless, stencil transformations represents a quite new research field, basically because, given multiple transformations, it is still unknown how to precisely measure their performance to select the best.

Also, the definition of *correct* stencil transformation is not so trivial; here a first notable work is represented by Meneghin's Ph.D. thesis [129], that formally characterized a wide set of stencils, and introduced several transformations, mainly

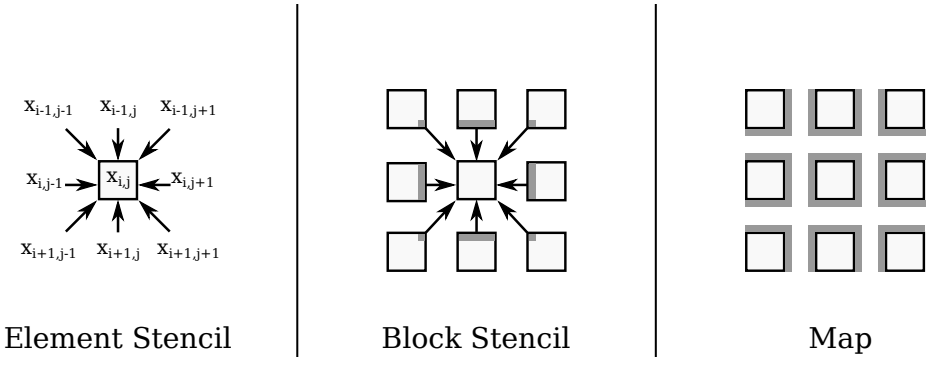


Figure 3.9: Transforming the mean filter in a Map.

to minimize data dependencies. However, the goal of a performance-driven stencil transformation is still an interesting research topic.

3.3 Expressing Parallel Paradigms

Up to this point, we defined the basic concept of structured parallel programming and introduced the most important pattern that characterize this approach. We did not focus, however, on *how* patterns are expressed. This is a sensible problem of this class of environment, because we should limit as much as possible the work of the application programmer, and be able to guarantee a set of patterns that allows a decent parallelization of most algorithms. Indeed, one of the most criticized point of structured parallel programming has always been the limited amount of patterns that were not able to capture many applications [61]. We introduce the first, and still considered state-of-the-art, approach based on the concept of algorithmic skeleton; then we present the approach, introduced by the ASSIST programming environment, which should overcome the main limitations of the skeleton-based approach.

3.3.1 Skeletons

The algorithmic skeletons defined by Cole[61] represent the first approach to structured parallel programming. He proposed 4 skeletons (*Fixed Degree Divide & Conquer*, *Iterative Combination*, *Cluster* and *Task Queue* - indeed a quite small, and particular, set of skeletons) obtained both by the isolation of particular algorithmic techniques, and by an analysis of patterns that could perform well on the initial target machine (a Transputer). From his idea, however, many researchers focused on finding general yet effective patterns that could be promoted to *skeleton*. Among the others, P^3L provided *pipeline*, *task farm*, *map* and *reduce*, plus *geometric*, *loop* and *tree* as data-parallel with stencils [18]; SKELib[68] offered only stream-based skeletons (*farm* and *pipe*), while Lithium [10] supported *pipe*, *map*, *farm* and *reduce*. Once stabilized, the set of used skeleton basically remained the same over the years: Skandium[113], one of the newest skeleton framework, implement *seq*, *pipe*, *farm*, *for*, *while*, *map*, *d&c*, *fork*, not introducing new patterns with respect to the first works.

All these systems employ the very same concepts introduced by Cole: the user just write a skeletal specification, such that a program is basically a composition of skeletons. The majority of environments defines three kinds of skeletons: data parallel, task parallel and sequential skeletons. Sequential skeletons encapsulate functions written in a sequential language and are not considered for parallel execution. The others provide typical task and data parallel patterns. For obvious performance reasons, data parallel skeletons can only encapsulate sequential skeletons: indeed there are no performance reasons to put a stream-parallel skeleton *inside* the calculation of a single element, because of the missing of a stream. Applications written in this

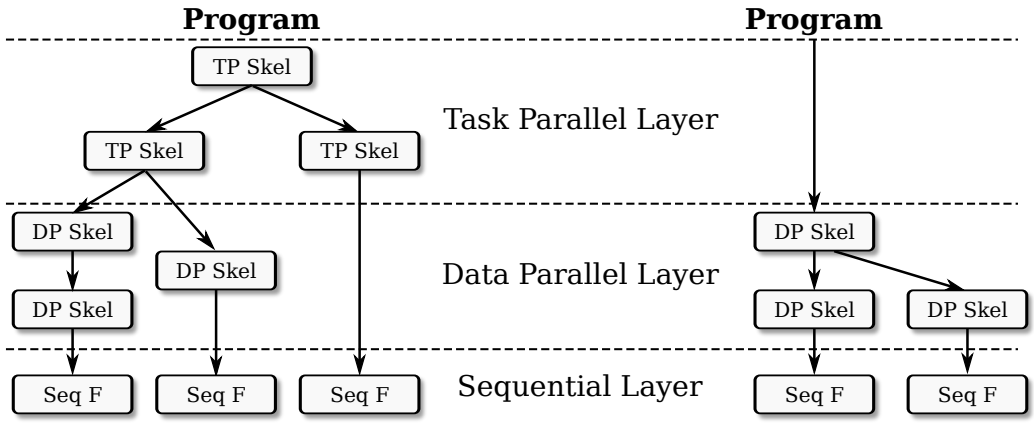


Figure 3.10: Three-tier structure examples of a skeleton-based program.

way respect the *three-tier structure* sketched out in Figure 3.10

The initial specification provided by the programmer may then be subjected to a cost-driven transformation process with the aim of improving the performance of the parallel program. Such transformation is done by means of semantic-preserving rewriting rules. A rich set of rewriting rules [11, 88, 87] and cost models [11, 151, 189] for various skeletons have been developed in the past.

3.3.2 ASSIST: Beyond the classical skeleton approach

Despite several advantages of skeletons, a strong evolution of structured parallel programming beyond such models is needed, at least for the following reasons:

a) in addition to the capability of expressing some typical parallel schemes, we need a larger degree of flexibility in expressing parallel and distributed program structures: we cannot afford to produce a skeleton for any data-parallel pattern, nor force the programmer to write applications respecting the 3-4 well studied patterns;

b) although very interesting, pattern composability is still limited: the three-tier structure of skeleton-based environments becomes a limitation when describing large, complex applications;

c) we recognize that parallel patterns cannot efficiently capture *every* parallel application: dynamic stencils, for example, cannot be modeled by a skeleton; we need to allow some kind of cooperation with different parallel environments so that skeleton-based patterns can cooperate with pre-existing, or manually optimized, parallel code.

Actually even Cole recognized this lack of expressiveness [61]:

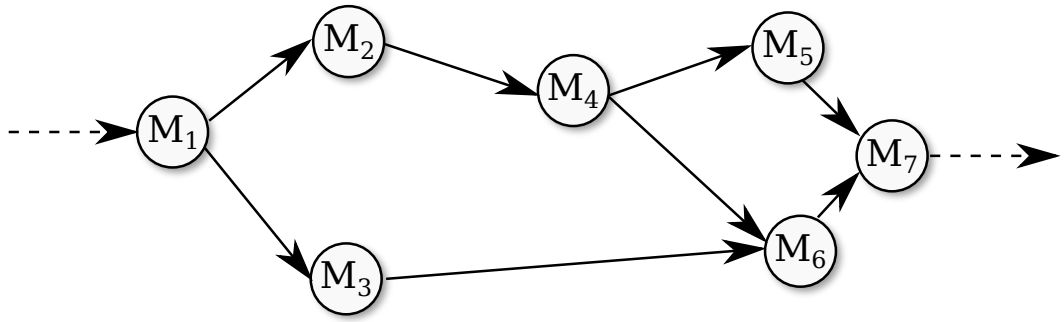


Figure 3.11: Example structure of an ASSIST program.

> Many parallel applications are not obviously expressible as instances of skeletons, whether existing or imagined. Some have phases which require the use of less structured interaction primitives. Some have conceptually layered parallelism, in which skeletal behavior at one layer controls the invocation of operations involving such ad-hoc parallelism within. It is clearly unrealistic to assume that skeletons can provide all the parallelism we need. We must construct our systems to allow the integration of skeletal and ad-hoc parallelism in a well defined way.

An interesting and effective approach to overcome the limitations of skeleton environments has been introduced by our research group with ASSIST [172] (A Software development System based upon Integrated Skeleton Technology). In ASSIST the common three-layered structure is replaced by a plain, graph-based structure: an application is described by a generic *graph* of *modules* connected by *streams*. This alone allows some basic stream-parallel paradigm such as pipelining, but at the same time permit very complex behaviors and loops among the modules that compose the application. Parallelism is also available *inside* the nodes, because each module represents a parallel pattern. An example of an ASSIST program is depicted in Figure 3.11.

ASSIST employ a novel approach to data-parallel by describing the parallel application (and its stencil) at the minimum partitioning level. This approach (called "Virtual Processor") generalize the class of data-parallel and allow the programmer to describe with a single formalism a generic data-parallel with a static stencil.

Lastly, a module is not forced to be implemented as a parallel pattern: the programmer may provide its specific, hand-made implementation of a parallel module. This effectively solves the cases in which a parallel paradigm cannot be applied.

3.3.3 The Virtual Processors approach

By following this approach we are able to provide a "generic" skeleton that can efficiently describe any kind of data-parallel program. The main idea is to describe

```
topology array [i:N][j:N] VP;

attribute long S[N][N] scatter S[*i][*j] onto VP[i][j];
attribute long L replicated;
```

Listing 3.1: Replicating and partitiong data structure in ASSIST.

```
virtual_processors {
  calc( in guard1 out out_matrix){
    VP i=1..N,j=1..N {
      for(h=0;h<N;h++){
        F(in L, S[i][h],S[h][i] out S[i][j]);
      }
    }
  }
}
```

Listing 3.2: Virtual Processor definition in ASSIST.

the application by using a set of "Virtual Processors" (VP), i.e. virtual entities that, like processors, owns a partition of the data structure, execute the calculation on it and exchange data with others.

The idea behind VPs is indeed quite simple: the programmer define the stencil at the minimum partitioning level by using this abstraction, while the parallel environment is in charge of analyzing it to determine if it represents one of the basic, well studied paradigm, or a new, "unknown" stencil. In any case, a proper *worker partitioning* must be established, so that the *Virtual Processors* becomes *Real Processors*, perhaps with a different stencil, and perform the computation.

A small ASSIST example is depicted in listings 3.1 and 3.2. The data structure is composed of a matrix of NxN elements and a single integer. The matrix is partitioned, so that each VP "own" a single element, and is in charge of computing it by following the *owner compute rule*. We can notice that the number of VP is well defined, and matches the elements of S. The second listing defines the "computation": each VP executes a function (F, that in ASSIST can be written in C, C++ or Fortran) that takes as input parameters the replicated value *plus* two elements that belong to different processors. Of course, the output value is only the element that belong to the VP, to respect the owner compute rule.

This way of describing data-parallels is indeed very powerful, because it explicitly define the stencil at the element level. From this, an intelligent compiler can apply all the *stencil transformation* described before, and optimize the stencil with respect to the execution environment.

As a side note we signal that, unfortunately, this optimization step was not available in ASSIST: the compiler grouped VPs in trivial ways, without any performance-driven optimization

3.4 Parallel patterns and their (many) implementations

A parallel paradigm describes the parallel entities and the structure of the interactions. However, the structure given by the paradigm is very general, so that there are many ways of coding it on a parallel machine. Although we can usually find a simple implementation that strictly resemble the definition of the parallel paradigm, there are many different versions that may perform better than the baseline, depending both on the algorithm and on the deployment architecture.

Let us take, as an example, the task farm paradigm. Even in this simple case (where each parallel entity is independent), there are many possible versions for its parallel implementation. The most common is the master/worker scheme [28], where a parallel entity (i.e. a process) called master is in charge of receiving tasks from the input stream, demanding work to a pool of workers and collecting results for the output stream. There are many cases, however, in which all this is too much for a single entity, which becomes the bottleneck of the entire application; we can therefore "split" the master in multiple entities. Depending on this splitting we obtain an emitter-worker-collector scheme [143], where we divide the dispatching of input data and the collection of results, or hierarchical masters [28] or even a mix of these two approaches. We can also be in the opposite case, in which the master is partially idle because faster than the workers; here we can let the master do some work on tasks in its idle time [143]. On top of this, we should also consider a dispatching technique that will limit as much as possible idle times in workers: on equally sized tasks a round-robin technique is usually good, but there are many cases in which tasks are unbalanced: because of the application, of the heterogeneous architecture, or others. Here comes an entire research area on task scheduling, with weighted scheduling, on-demand scheduling, task stealing and so on.

If this is the case of a simple pattern (in which cooperation is done only between each worker and the supporting entities), it becomes clear that in the case of more complex interactions, like with data-parallel programs, the number of possible choices, and optimization possibilities, further increase, making the selection of the proper implementation a considerably hard task, even for an automatic tool.

3.5 Mastering the possibilities, one piece at a time

We started this chapter with a strong, important point: the need for a programming environment that allow performance portability. In the previous sections we understood that Structured Parallel Programming *can* be a solution for the problem:

- we have a small, pre-determined, set of stream-parallel patterns;

- we have an entire family of data-parallel patterns, and a powerful model (the "Virtual Processor" approach) to describe them;
- we have a set of stencil transformation, that allows us to transform the data-parallel code;
- we have a large number of implementation choices for each paradigm, which can fit one hardware architecture better than another.

What we still do not have, instead, are performance studies of structured parallel programming on multi-core that could:

- give us some insight on which of the many choices offer good performances on multi-core architectures;
- tell us if multi-core are just to be treated as shared-memory architectures, or if there can be specific multicore-related optimization on parallel patterns;
- help us in finding ways to efficiently exploit specific hardware facilities that are emerging in the last multi- and many-core, such as multithreading, explicit message passing among processors, etc.

Finally, but, most important, we need *some way to compare the possible choices at compile-time*, so that the compiler can choose a good implementation (preferably the best) according to the previous points.

With this thesis we start filling this gap, mainly by means of specific performance tests of explicit multicore-related optimizations and the definition of the architectural performance model for a specific multi-core architecture that, tied together with generic pattern-based cost models, will allow us to evaluate different implementations and determine, for each specific program, which solution will offer better performances.

All these studies will be driven by keeping in mind our long-term research objective: an innovative programming environment based on structured parallelism, able to truly provide *parallel portability*.

3.6 Towards a novel parallel programming environment

The long-term project of our research group is ASSISTANT, the extension and adaptation of ASSIST for the current world of parallel computing, composed of multi-cores, pervasive grids and clouds. Many of the principles introduced in ASSIST are *inherited* and *extended*, in order to provide a significant leap forward in the world of multicore-oriented parallel programming.

Respecting the basic ASSIST principles, a parallel program will be described as a generic graph of stream-connected parallel modules. Each module will be constituted by one of the previously mentioned parallel patterns, or by a VP-based description in the case of a data-parallel. As in ASSIST, the programmer will be able to write the algorithm code by mean of the most used sequential language (C, C++, Matlab, Java, and so on).

Programming models based on libraries are considered unsuitable for achieving the desired level of programmability and performance portability: our environment will need an intelligent *source-to-source* parallel compiler, able to analyze the module-based description to determine the possible parallel implementations, evaluate them for the target machine and, finally, produce the source code of a low-level parallel program.

Our experience in parallel programming also pointed that there are many cases in which performance portability is not *completely* achievable at compile-time: the cost model may be not detailed enough to accurately fit the <application, implementation, architecture> tuple, or some model parameters may be unpredictable (because of both the architecture and the algorithm) so that a mere compiler-based performance portability becomes ineffective. To handle all these important cases, it is also mandatory to support *adaptivity*, by means of efficient run-time reconfigurations, in addition to static optimizations[32].

In addition, to better allow performance portability and adaptivity, we believe it is necessary to allow the programmer to *explicity* define different patterns for each module. This way, if multiple parallel patterns (with different performance characteristics) are known by the programmer, we further increase the possibilities of our compiler. This approach, whichd has been introduced in our works with pervasive grids[32], remains coherent with the programming model: we are adding to the set of compiler-defined transformations (that represent different implementations of a module) others, not automatically derivable, transformations. The compiler will then use its cost model to select (and optimize) the best among the whole set of implementations.

The resulting "compilation workflow" is depicted in Figure 3.12. Of course, the meaning of compilation now is stretched to the whole execution because of the run-time-based reconfigurations. We can easily notice how important is the *Cost Model*, which affects basically every step of the workflow, making it a first-class citizen in our approach.

In the first phase, we use the cost model to determine which modules of the application graph negatively affects the global performance: of course it may happen that, of the many parallel modules, some are so fast that does not truly require a parallel implementation; moreover, considering the *finite set of resources*, a proper balancing of parallelism is required among the various modules, to obtain the maximum performance.

In the second step the compiler determine, for each parallelized module, the best parallel paradigm and its implementation, considering both the user-provided and

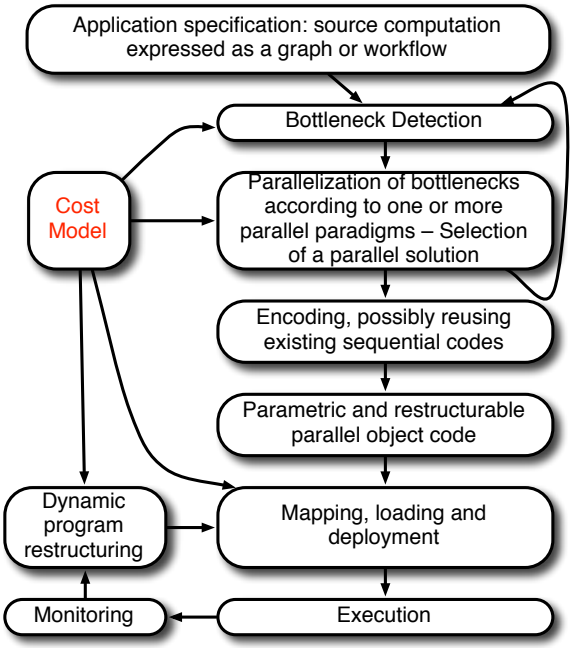


Figure 3.12: The "compilation workflow" in our programming environment.

the automatically derived transformations. At the end of this step the bottleneck detection is run again, considering the expected implementation of each module and their specific cost model. If new bottlenecks are found, these two steps are executed iteratively to further refine the parallel implementation.

The compiler then generates the low-level parallel source code, to be compiled using a generic C/C++ compiler (such as gcc or icc). The resulting application, however, is also enriched with monitoring tools and other possible parallel implementations, so that, by continuously monitoring and applying the cost model, the program self-adapt to better match the running environment and guarantee the best possible performance.

3.7 Target architectures

Given the high amount of tests and experimentation in this thesis, we will briefly introduce (and comment) once and for all, the multi-core architectures that will be used in our tests.

Although there are many different multi-cores today, and probably there will be much more in the future, we were able to select a representative group of architectures from the ones discussed in Chapter 2. Given the almost exclusive presence of x86-based processors in general purpose multi-core architectures, we selected three architectures, differing on the processor implementation, the memory hierarchy and

the number of cores. We also used multi-chip servers, to reach interesting numbers of cores. Finally, as an "insight" of future multi-core architectures, we selected one of the larger (in term of cores) multi-core available: the Tilera Tile*Pro*64. To summarize, the platforms are the following:

- two Intel Xeon® E5-2650, Sandy-Bridge based architectures, composed of 8 cores, 20 MB of shared cache and a memory controller per chip, running at 2.00GHz;
- four Intel Xeon® E7-4820, Westmere-EX based architectures, composed of 8 cores, 18MB of shared cache and a memory controller per chip, running at 2.00GHz;
- two AMD Opteron™ 6176, for a total of 4 Magny-Cours based chips, composed of 6 cores, 6MB of shared cache and a memory controller per chip, running at 2.3GHz;
- a single Tilera TILEPro64™, composed of 64 cores, no shared cache but 4 memory controllers, running at 866MHz.

We can notice that all the three x86 architectures basically employ the same conceptual characteristics (i.e. multiple chips connected by point-to-point links, chips with a small number of cores, a memory controller and a large shared cache), and differs only for technical implementations; despite that, we will see very different results in the aspects treated in this thesis. The many-core architecture, instead, is notably different from the others, pointing up the fact that the previous conceptual organization may not be really feasible with a large number of cores.

The Tile*Pro*64 chip is very interesting from many points, because employs a large number of specific features, like explicit on-chip message passing, controllable cache coherence, multiple memory interfaces, and an on-chip Ethernet interface. Finally, the programming environment is complete and exhaustive: almost every detail of the architecture is extensively described in the documentation, and a clock-accurate simulator is provided.

For these many reasons we elected the Tile*Pro*64 as the *reference architecture* for this thesis: many results are generalized on the other architectures, but the detailed performance model was explicitly parameterized for this chip, and extensively verified by using both the real architecture and the clock-accurate simulator.

Part II

Cost Models

Chapter 4

A hardware-dependent model based on Queueing Networks

Given the considerable different characteristics of each multiprocessor, the emergence of parallel architectures emphasized the need of some way to evaluate a program without having to implement and tune it. The study of formal approaches to the process of analyzing the performance of computer systems has always been of great interest in computer science, and quickly become very important in parallel computing. It is very interesting to note that performance evaluation is effectively used in many aspects of computer science, starting from hardware development, up to software design.

Performance evaluation has been (and still is) widely applied in hardware designing [3, 36, 101, 116, 130, 133]: given the complexity of the projects, it is difficult to evaluate the impact of a change, and very expensive to test the changes by means of prototypes. For this reasons engineers commonly use performance evaluation tools to drive the design of processors and parallel machines, in order to better balance the performance/cost ratio, and to produce architectures with specific performance requirements.

Nevertheless, performance evaluation is also quite used in software development [2, 23, 108], in all those cases in which programs need to satisfy specific constraints (not only performance-based but, for example, memory- or power-based), such as real-time system, or in general to predict specific aspect of the to-be-developed software.

Performance evaluation is usually achieved by using three different techniques: **measurement**, **simulation** and **analytical modeling**. All these techniques play equally important roles in performance studies, because each one has its own advantages and disadvantages, that basically consist in a proper mix of *precision* and *cost*. We cannot really say that one technique is *always* better than another, as it usually depends on what we are evaluating. When a single technique cannot be effectively applied, a mix of the three is used to effectively evaluate the performance of the system.

Our final goal is to predict the performance of a specific parallel implementation of a program. It is important to note that the performance models presented in Chapter 2 were defined with a different objective, which is to evaluate the asymptotic characteristics of a parallel program, *independently* of the running architecture. It is no coincidence that these models evaluate algorithms on abstract, simplistic, architectures. However, in our case, the general characteristics are already known because of the use of parallel patterns. Given our need to evaluate implementations that may differs for very small details, we also need *detailed* performance models, to find the best one.

We stressed in the previous chapters that the performance of an implementation *strictly* depend on both software and hardware characteristics. We know that performance evaluation is successfully applied to predict the performance of those two systems *separately*. In this thesis we *merge* the two approaches, in order to evaluate the couple <program,architecture> and therefore ending with a prediction of the parallel implementation we are analyzing.

An interesting point of this approach is that, given the generality of the performance modeling techniques, it is easy to describe the whole system (comprised of hardware and software models) by:

a) using a single methodology,

b) verifying our modeling intuitions with previous works, and

c) combining already available models of small parts of the system.

Because of these points our work has been enormously facilitated, so that we will be able, at the end of the thesis, to provide some interesting modeling results and prove its precision w.r.t real programs running on a real architecture.

The architecture-dependent nature of our models poses a huge problem with respect to our need of a generally applicable performance prediction method. In other words, we want to provide a general model suitable for the generic class of parallel patterns running on a class parallel architectures, but at the same time we recognize that the model need to take into account of the single characteristics of the specific architecture. This may seems, and in a certain way is, a contradiction: the model should be *general*, to be able to represent any architecture, but also *detailed*, to precisely predict the performance. This is probably not really achievable by using a single model.

In this chapter, we will present a *general approach*, a methodology to *derive a model* to match the details of a specific architecture and of the parallel pattern. The approach will then be used in the following chapters, targeting both specific architectures and specific parallel pattern implementations.

The ideas presented in this chapter have been studied and developed in our research groups for many years, having their roots in the P^3L skeleton language and its implementation templates; however, many of the concepts presented were yet to

be published and during this thesis were further refined. The most comprehensive work that address the methodology is the Italian textbook [173], where it is applied to an abstract multiprocessor architecture, thus simplifying the model and its parameters to a level understandable by students. For these reasons this chapter does not consist in a novel contribution of the thesis but, because of the limited visibility of the approach, we decided to present it in a specific chapter.

4.1 A general approach to parallel performance prediction

In our programming environment, a parallel program is defined as a graph of cooperating (parallel) modules. This means that the programmer have two ways to express parallelism:

- **Intra-module parallelism**: each module is described by a parallel pattern, and therefore able to exploit stream-parallel or data-parallel parallelism, depending on the chosen implementation;
- **Inter-module parallelism**: modules are composed in computation graphs with a general structure, where interactions are possible by means of *streams*, effectively adding a second layer of stream-parallel parallelism.

The methodology that we are introducing is aimed to completely model the performance at any level, analyzing both the internal behavior of a single module, and the performance of the entire computation graph, by providing a performance modeling approach expressed in terms of fundamental results in the area of Queueing Theory and Queueing Networks. In this way we will be able to formalize important issues related to:

- how to evaluate the performance of a graph computation starting from the knowledge of the performance of each module;
- how to evaluate the effective performance of a module based on the ideal performance behavior of all the modules of the computation graph;
- how to detect bottlenecks in a computation graph, that is modules that seriously limit the performance of the entire application.

This methodology is not new, and has been deeply described in [125], so we will just introduce the concept, needed to intuitively understand the ideas and how the model work; the interested reader can refer to [125] for more specific details.

The basic idea consists in modeling the performance of a module M (either sequential or internally parallel) by abstracting its behavior as a queueing system, as shown in Figure 4.1. This scheme is a logical one, not necessarily corresponding

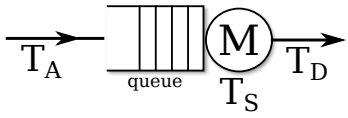


Figure 4.1: A computation module modeled as a queueing system.

to the real implementation. However, it is aimed at capturing the essential elements of the problem at hand. The behavior of a queueing system is characterized by expressing five different parameters:

1. the service discipline: if not explicitly defined the FIFO policy is assumed;
2. the queue size, that is the number of buffer positions available for storing the incoming requests to the module;
3. the probability distribution of a random variable *inter-arrival time* t_a (i.e. the time interval between two consecutive arrivals of requests), with average value T_A and (optionally) variance σ_A;
4. the probability distribution of a random variable (ideal) service time t_s, which represents the ideal time needed to serve a customer, i.e. the time passed between the beginning of the executions on two consecutive stream elements. We denote with T_S and σ_S the average value and (optionally) the variance of this random variable;
5. the probability distribution of a random variable *inter-departure time* t_d (with average value T_D and optionally variance σ_D), which indicates the time between two successive result departures from the module.

Each computation module can be abstracted as a queueing system and the computation graph can be described as a network of queues [111], where the departures of some nodes form the arrivals of others. From the network topology viewpoint queueing networks can be categorized into two broad classes namely **open queueing networks** and **closed queueing networks**. In an open queueing network a possibly infinite number of requests are generated by source nodes, go through several nodes or even revisit a particular node more than once and finally leave the system. On the other hand, in a closed queueing network requests neither arrive at nor depart from the system, but a fixed number of requests continuously circulate through the nodes of the network.

In our case, the graph of modules depict an **open queueing network**, given the presence of infinite streams. For the sake of simplicity our approach will be limited to **acyclic computation graphs**, where each task follows a certain path, passing through each modules *at most once*.

With this simplification, we are able to analyze the performance of this kind of graphs in a completely independent way w.r.t the internal behavior of each computation module, i.e. it may implement any parallel pattern. The only parameter required is the average value of the ideal service time of each module. The behavior of intra-module parallelism can be treated independently, and will be analyzed in Section 4.2.

The evaluation methodology - derived from common queueing theory - consist in two interrelated phases:

- **Transient analysis** consists in a study of the network behavior in the initial transient phase of the execution. For transient phase we intend the initial situation in which the performance behavior of each node can significantly change in relatively short time periods due to the starting conditions of the network (e.g. due to the size of the queues);

- **Steady-state analysis** provides results for evaluating the effective performance (i.e. the mean inter-departure times) of each node in the network *after* the transient phase, when the performance behavior of each module is completely stabilized and it is no longer influenced by the initial conditions.

Being interested to the performance behavior of a stream-based application, we consider the transient analysis *irrelevant*, because of the length of the streams, that render this phase very small w.r.t the global computation, and focus on the **steady-state analysis**.

The steady state analysis is typical in queueing networks and can be obtained by exploiting several methods, such as using product form, mean value analysis or simulations. The result of a steady state analysis are a set of performance indexes for each queue of the system, including the throughput of a queue, that represent the inverse of what we defined as "interdeparture time".

In particular, referring to the compiler workflow already depicted in Chapter 3 (in Figure 4.2), we first have to transform the application graph in a queueing network, to be able to perform the steady-state analysis. Each module is naturally described by a queue, but we need to evaluate some parameters, in particular:

1. The inter-arrival times for each *external input stream* (i.e. the streams used by the program to receive data);

2. The routing probabilities, when a module has multiple output stream;

3. The service time of each module.

The resulting transformation is exemplified in Figure 4.3

Theoretically we would also need to know the probability distribution of inter-arrival and service times, to fully represent the system as a queueing network. However, [125] present a simple methodology that only require the mean values of these

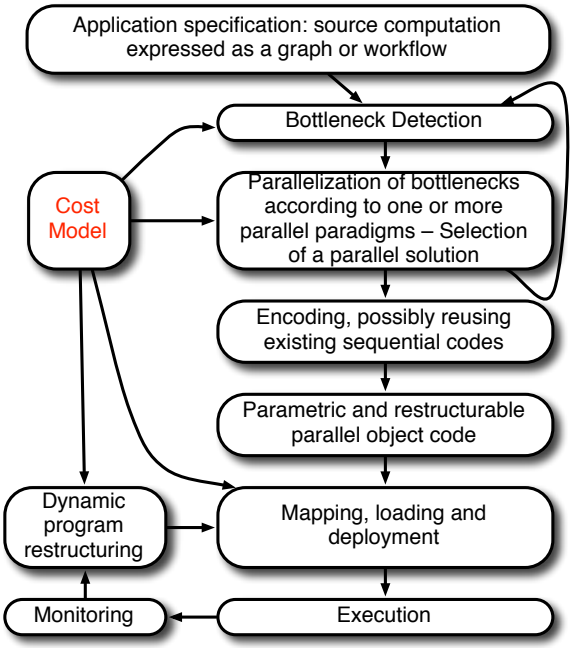


Figure 4.2: The "compilation work-flow" in our programming environment.

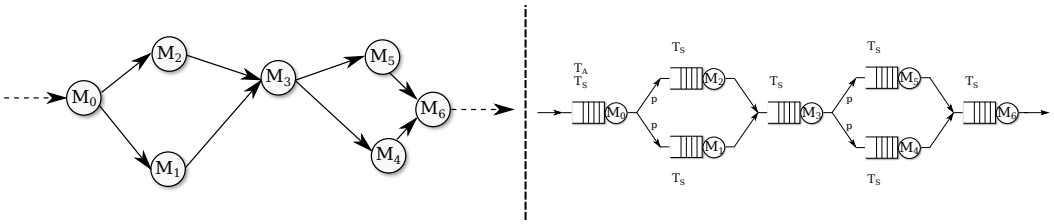


Figure 4.3: An example module graph and its queueing network representation.

parameters, enormously simplifying our model. The interested reader can refer to [125] for a complete description of this methods and its applicability limits. In general we can use of any classic method of queueing theory (using exponential probability distributions) when the previous approach is not feasible.

Of the three parameters, the first two must be expressed by the programmer, as they depend either on external factors (i.e. the inter-arrival times) or on properties unknown by the programming environment, so that the compiler has no mean to estimate them. On the contrary, the third represents the ideal performance of each module: it depends on the parallel implementation chosen by the compiler, so specific performance models will be used to derive those values. For the moment, we assume these are automatically obtained in some ways. Later in the chapter, and through the whole thesis, we will deal with this problem.

Back to the compiler workflow depicted in Figure 4.2, we can notice the iterative approach of steps 2 and 3, that are executed until we find the best solution. To start this iterative approach, we need a simple, pre-defined configuration of the parallel program. We consider each *module* implemented by a sequential version, so that we use a processor per module. With this configuration we can start evaluating the parallel program (step 1), by:

a) fixing the source stream mean inter-arrival time and the routing probabilities by using the programmer-given values;

b) evaluating the ideal **sequential service time** of each module (that is the mean execution time of a task of a sequential implementation);

c) computing the steady-state result of the queueing network.

This way we are able to find the *bottleneck* of the application, i.e. the module M that, in this configuration, slows down the entire application. We are able to do this by using an interesting property of the steady-state analysis[125]:

Proposition 4.1. *(Steady-state behavior of a node). At steady-state the effective interarrival time of each node is equal to its inter-departure time. If that inter-arrival time also coincides with the ideal service time of the node, the node is a bottleneck, otherwise the node is not a bottleneck.*

An example of the end of steps b) and c) is depicted in Figures 4.4 and 4.5, respectively. We can easily notice that the only node in which $T_A = T_S = T_D$ is node 2: the corresponding module is the bottleneck of our example application.

It is also important to note that, for a given configuration, *a single bottleneck* exists, i.e. all the other modules will slow down to respect its computation time. However, once the bottleneck is removed, *a new bottleneck may emerge*, so the iterative approach is required to reach the optimum parallel configuration for the graph.

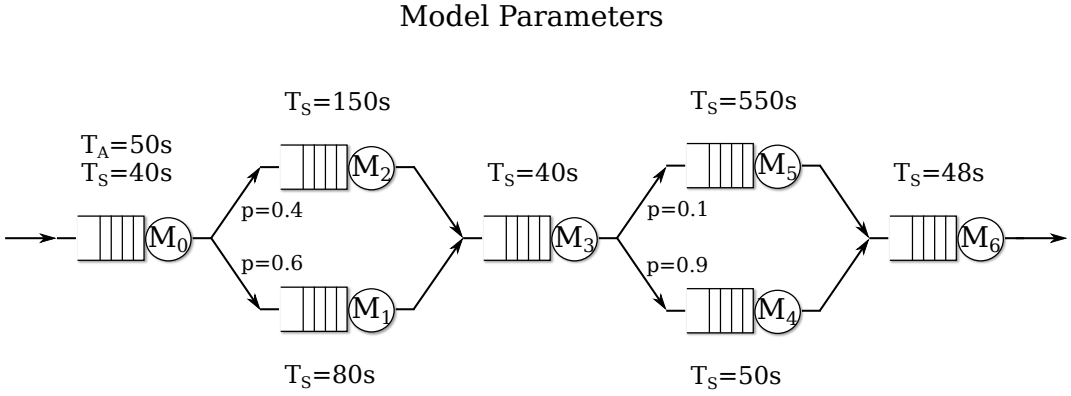


Figure 4.4: The fully parameterized QN-based model.

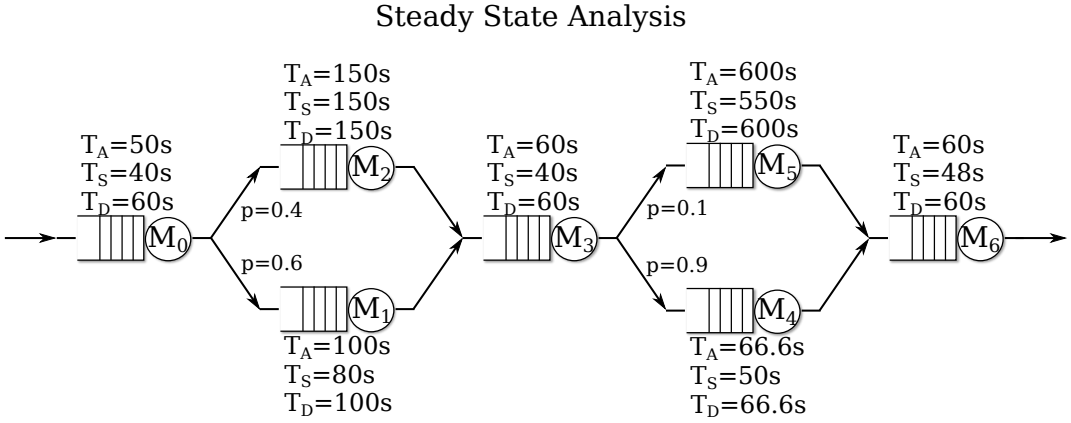


Figure 4.5: The result of the steady-state analysis of the model.

From this point we diverge from the approach presented in [125], because that work uses an approximate performance model for the modules, considering the ideal scalability, in which using a parallelism degree of n offer a service time n times better than the sequential one:

$$T_S^{(n)} = \frac{T_S^{(1)}}{n}$$

with this approximation, it is always possible to remove a bottleneck (assuming the correct amount of nodes is available). For this reason the approach in [125] removes the bottlenecks *during* the steady state analysis.

In our case, given the use of a detailed performance model for each possible module implementation, the estimation of $T_S^{(n)}$ is more complex: in general the scalability of a module does not coincide with the previous one and, at some point, may stop.

We also consider that in common applications we do not have an infinite number of nodes, and that it is also usually important to reduce, as much as possible, the number of used nodes for power consumption reasons.

For this reason we propose a different approach, still based on the one in [125], in which we try to remove *one bottleneck at a time*. We can select the most limiting module of the application, and *try* to remove the bottleneck. If the bottleneck is removed, we can re-analyze the graph, ending with:

1. no more bottlenecks in the graph: the input throughput is sustained, so our application does not need further refinements, or

2. a new bottleneck in the graph: removing the first bottleneck exposed a new module that was limiting (to a lesser extent) the performance of the application; we can now remove this bottleneck, re-evaluate the graph once more and continue this iterative approach as long as we have no more bottlenecks.

However, considering the module performance model, we also may end with not removing a bottleneck. In this case, no further steps are required, as even by parallelizing the other modules we will not improve the performance of the whole graph.

The notable advantage of this approach w.r.t [125] is that we are able to minimize the overall parallelism degree (i.e. the number of nodes used by the whole application) by parallelizing one module at a time. With the previous approach, we may parallelize modules that would have become bottlenecks after some iteration of our approach. However, it may be not possible to remove one of the bigger bottlenecks, thus ending in parallelizing modules that does not hurt the performance of the application.

Removing a bottleneck

Once a bottleneck is found, the module will be parallelized, as in step 3 of the compilation workflow, by following this approach:

1. **Determine the required service time**: first of all, we need to calculate which average service time of the selected module will remove the bottleneck. For this we need the *ideal inter-arrival time of* M, i.e. the inter-arrival time of the system where M is not slowing down the other modules. This time can be obtained by setting the service time of M to 0 and re-apply the steady state analysis. Given its new inter-arrival time T_A^I, M is not a bottleneck if

$$T_S \leq T_A^I \tag{4.1}$$

   For example, in the model of Fig. 4.4 example, we have $T_S \leq 137.5s$.

2. **Determine the required parallelism degree and the correct implementation**: given the unknown properties of the service time function $T_S^{(x)}$ of this module, *in theory* we should try each possible parallelism degree to find (if exists) a solution that removes the bottleneck. Moreover, we are interested in a solution that minimizes the number of nodes for power consumption reasons. The estimation of each point of $T_S^{(x)}$ is not feasible, so we need to cut down the space of possibilities. For this we adopt a reasonable heuristic that allow us to limit the number of evaluations of $T_S^{(x)}$. We start by considering that, in practically all the cases, a module does not *superscale*, so that we can find a lower limit to $T_S^{(x)}$ by using the function

$$T_S^{(x)} \geq \frac{T_S^{(1)}}{x}$$

   If this hold, we can estimate the *minimum* number of nodes required to achieve the required service time:

$$n \geq \frac{T_S^{(1)}}{T_A^T}$$

Therefore, this represent the starting point to evaluate $T_S^{(x)}$ for our module. In fact, having the (obvious) requirement of selecting an *integer* value of n, the starting point will be

$$n = \left\lceil \frac{T_S^{(1)}}{T_A^T} \right\rceil$$

We also apply a higher limitation to n by using the maximum allowed number of nodes.

$$n \in \left[\left\lceil \frac{T_S^{(1)}}{T_A^T} \right\rceil, n_{max} \right]$$

If we are lucky, our module scale well and the first possible value of n already represent the correct parallelism degree required to remove the bottleneck; if it is not, we know that we will need to find an higher value of n that removes the bottleneck in that interval.

Given that $T_S^{(x)}$ is a generic function, any point of the interval is a is possible candidate to remove the bottleneck. However, in most cases $T_S^{(x)}$ is, or can be approximated with, a *monotonic* function. We therefore apply a second heuristic, assuming that the $T_S^{(x)}$ function is monotonically decreasing, i.e. it cannot increase by rising the number of nodes. If this hold, we can apply some optimized search function in the selected interval, for example a bisection method, to effectively reduce the amount of analyzed points of $T_S^{(x)}$ to find the best value n that, however, *may not* remove the bottleneck.

After these steps we removed (or limited in those cases in which a complete removal was not possible) the bottleneck; now we need to re-evaluate the steady-state analysis of the graph: indeed removing the bottleneck drastically change the performance of the graph, resulting in three possible cases:

1. **A new bottleneck is found**, so that there exist a new module K such that $T_A = T_S$: we proceed parallelizing K; or

2. **The bottleneck has not been removed**: in this case we can stop our evaluation, as there is no way of further increasing the performance of the graph; or

3. **All the bottlenecks were removed**, so that all the modules satisfy the condition $T_A > T_S$: the performance evaluation step is done.

For our example, let's assume that we are able to implement the module with a *data-parallel* approach that will achieve $T_S^{(2)} = 85s$ and a *farm* approach with $T_S^{(2)} = 80s$. Now, they all introduce a slight overhead (a perfect scalability will produce $T_S^{(2)} = 150/2 = 75s$), and the data parallel is even a bit slower; however they both remove the bottleneck because of the integer approximation of n, so we choose the

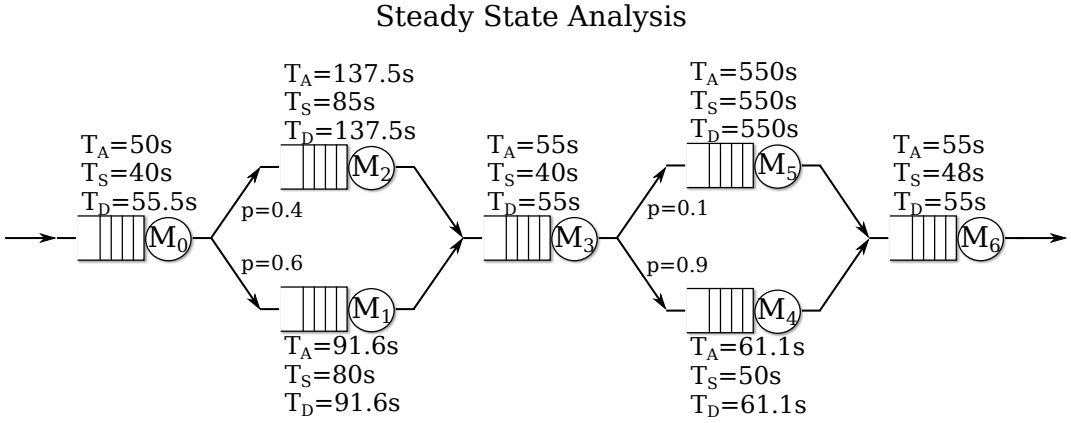


Figure 4.6: Result of the steady-state analysis after the parallelization of module 2.

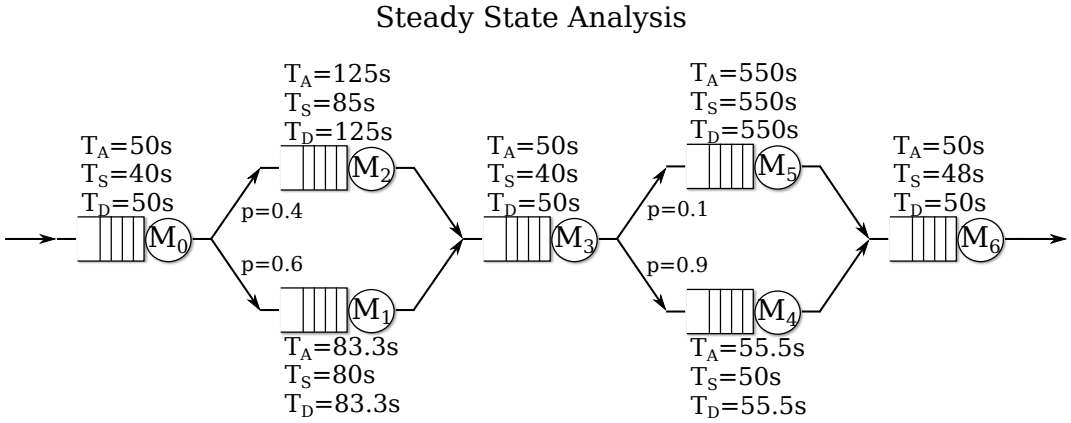


Figure 4.7: Result of the steady-state analysis after the parallelization of module 5.

data-parallel approach for the possible benefits w.r.t latency. We effectively removed the bottleneck, but a steady-state analysis of the new model (Figure 4.6) shows that a new bottleneck exists: module 5 is now slowing the application. However, after this second iteration, all bottlenecks are removed as shown in Figure 4.7.

It is worth noting that in a general iterative process, a module M may be analyzed more than once, depending on the complexity of interaction of the graph; nevertheless, the process will definitely end at a certain point where we will either:

- reach a steady-state solution with no bottlenecks, or
- reach a steady-state solution with unsolvable bottlenecks.

4.1.1 The case of single-element streams

The presented analysis works very well in the case of stream-connected modules. However, in our programming environment, we also want to allow the programmer

to work on pure data-parallel parallelizations, i.e. parallel programs that works in the presence of a single, possibly large, data structure. From a modeling point of view, this is possible by:

a) expressing data parallel parallelizations of a module, and

b) connecting multiple data-parallel modules with single-element streams in the case of complex parallelizations.

Nevertheless, the stream-based analysis does not hold anymore, as we are indeed interested in different metrics, i.e. minimize the time *the single element* spend in the whole system.

We propose a latency-based performance evaluation, in which the user provide a special value for source streams: $T_A = \infty$. This basically represent the fact that no further elements will be received from the source stream. Recognizing this particular value, the compiler will perform a different performance analysis.

Given the absence of an input frequency, we do not have an intrinsic limit on the performance of our application: we can lower the latency as much as possible, considering only the limitation on the maximum parallelism degree, given by the amount of nodes.

In addition, we do not have to partition the amount of nodes on the different modules, as each one will work in different times (because we do not have multiple elements).

The problem is thus limited in finding, for each module, the best configuration, given the maximum amount of nodes. This represent, of course, a much simpler problem w.r.t the previous one.

4.2 Performance prediction of a parallel module

In the previous section we discussed the performance prediction of a parallel application, assuming we were able to predict the performance of each module.

In this section we discuss the general methodology to evaluate the modules, and provide a simple example to clarify its application. Because of the deep implications of the implementation in the performance model of a module, at this point we are unable to provide the model of each possible pattern, so we can only present the underlying methodology. In the following chapters, however, when optimizations are presented, will discuss the possible ways to model these features and, in the last chapter of the thesis, we will show the application of this methodology to a real-world example.

The basis of our methodology can be considered, in some way, inherited from *BSP* or *LogP*, in which we identify and measure, for each processor:

a) a **sequential code** that is run independently from the others;

b) **synchronization points**, in which the processor waits for others to complete their tasks;

c) **communication points**, in which the processor sends and receives data to or from the others.

However, with respect to the previously presented models, we aim to a precise evaluation, based on the program and the architecture, of both the *service time* and the *latency* of the module.

The main idea that allows us to tackle the complexity of the model is that we are able to *separate* the behavior of the algorithm and that of the parallel pattern.

The sequential code is, indeed, strictly dependent on the application, given by the user, and does not depend in any way on the parallel implementation of the pattern, so it can be estimated from the code inserted in the patterns.

The other two points (synchronizations and communications) are not influenced by the sequential code, as they are defined by the pattern and its implementation; moreover, because of its "pattern-based" nature, each implementation will have well defined communication and synchronization points, allowing us to study these two points on a per-pattern implementation basis.

This way we are able to restrict the scope, and therefore enormously simplifying the development of a performance model, w.r.t. an undisciplined parallel programming environment, in which communications and synchronization are defined by the application programmer in a generic way.

4.2.1 An example: cost model for a trivial task-farm implementation

Here we introduce a simple implementation of a task-farm, that will be used in the rest of the chapter to exemplify the methodology used to define a model for a pattern implementation. We selected the farm as a starting point because of its simplicity; nevertheless, it still represents a significant example to understand the underlying methodology. The discussed implementation is the classic, skeleton-based implementation of P^3L [140, 143], initially defined for distributed memory architectures, with no particular optimizations w.r.t multi-core architectures. The pattern uses the Emitter $\rightarrow$ Workers $\rightarrow$ Collector scheme, exploiting two cores to implement the supporting entities. The pseudo-code of the three classes of processes is reported in Listings 4.1, 4.2 and 4.3.

Regardless the existence of a hardware shared memory, *this* implementation uses explicit communications, implemented with a simple message-passing library. The pseudo-code is pretty straightforward: the emitter follows a round-robin scheduling policy, starting from the first worker ($i = 0$). The collector follows a similar pattern, receiving data from the workers (again, in a round-robin fashion) and forwarding them to the output stream. For the sake of simplicity, this example do not introduce the *termination code*, needed to identify the end of the stream and to close

```
while(true){
  for(i:0..n-1){
    message temp;
    receive(temp, input_stream);
    send(temp, worker[i]);
  }
}
```

Listing 4.1: Pseudocode of the emitter.

```
while(true){
  for(i:0..n-1){
    message temp;
    receive(temp, worker[i]);
    send(temp, output_stream);
  }
}
```

Listing 4.2: Pseudocode of the collector.

```
while(true){
  message temp;
  receive(temp, emitter);
  ...
  // Computation
  ...
  send(temp, collector);
}
```

Listing 4.3: Pseudocode of the generic worker i.

the application. Communications are **asynchronous** and **blocking**, following a semantic similar to that of ECSP[20] and, w.r.t MPI, analogous to the `MPI_Send` and `MPI_Recv`, but with an important difference: **asynchrony is guaranteed** for a fixed number k of messages, so that the send operation do not need to wait for the corresponding receive, regardless of the length of the message.

Despite the lack of many information about the actual implementation of the pattern, we can already start building a performance model for our task-farm. In particular, we already know the **sequential code** of each process of the application and the flow of tasks through the entities of the computation.

4.2.2 Sequential code analysis

We identify three classes of processes, each one executing a different code; we can start by assigning an average time length to each step of the code. In particular, for each task

- The **emitter E**:
  1. receive the task from the input stream (T_{E-recv});
  2. send the task to the selected worker (T_{E-send}).
- The **worker W**:
  1. receive the task from the emitter (T_{W-recv});

2. compute the task (T_{W-calc});
3. send the computed task to the collector (T_{W-send}).

- The **collector C**:

1. receive the computed task from a worker (T_{C-recv});
2. send the computed task to the output stream (T_{C-send}).

Assume, for now, that we are able to measure a mean value for all these parameters, and that all the receive and send times do not account for blocking (i.e. when we execute a receive, the data to be received has been already sent, and when we execute a send, we are able to send the data). In this condition, we have that T_{*-send} and T_{*-recv} contain only the time needed to exchange the message (i.e. access to the shared structure that represent the communication channel, modify it and write/read the message). T_{W-calc}, on the other hand, represent the time to execute the computation on a single element.

4.2.3 Latency Model

By using the information obtained by the previous analysis we can easily evaluate the *latency* of the parallel pattern. In particular, we can see in Fig. 4.8 the temporal behavior of our pattern in the presence of a single stream element (a perfect candidate for studying the latency). In practice, not only each process executes its part sequentially, but *the whole computation* is executed sequentially, so that we pay the whole computation time T_{W-calc} *plus* all the communications. From the figure

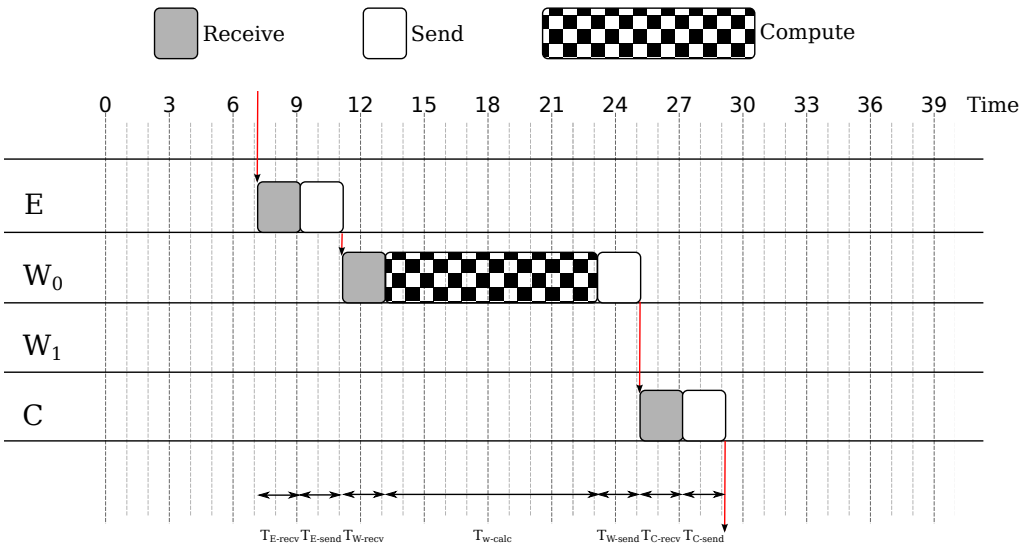


Figure 4.8: The temporal behavior of the farm implementation with a single element.

we can easily produce a formula to predict the latency of this implementation, that consist in the sum of the latency of each entity of the parallel pattern:

$$\begin{aligned} L_{FARM} &= L_E + L_W + L_C && (4.2) \\ L_{FARM} &= T_{E-recv} + T_{E-send} + \\ &+ T_{W-recv} + T_{W-calc} + T_{W-send} + \\ &+ T_{C-recv} + T_{C-send} && (4.3) \end{aligned}$$

From this we can already understand the problems of a stream-parallel pattern in the presence of a single element: despite the number of processes and cores allocated, *the application is not parallel.*

4.2.4 Service Time Model

Of course, with a stream things are different, as exemplified in Fig. 4.9 and 4.10.

The two cases differ on the ability of the emitter to *sustain* the computation. In particular, in Fig. 4.9 we can easily see that each worker can start the next task as soon as it finishes the first. The Emitter, on the other hand, after a transition phase, *stop* after each send, to wait the completion of the previous receive from the worker. This is because the channel implement an asynchrony level of 1: it allows only one message in the buffer. This example was selected to keep the drawing small, but the behavior remain the same with different k: by using a higher asynchrony we just increase the length of the transition phase, but the steady-state behavior of the pattern will remain the same. Fig. 4.10 shows the opposite, i.e. a case where the

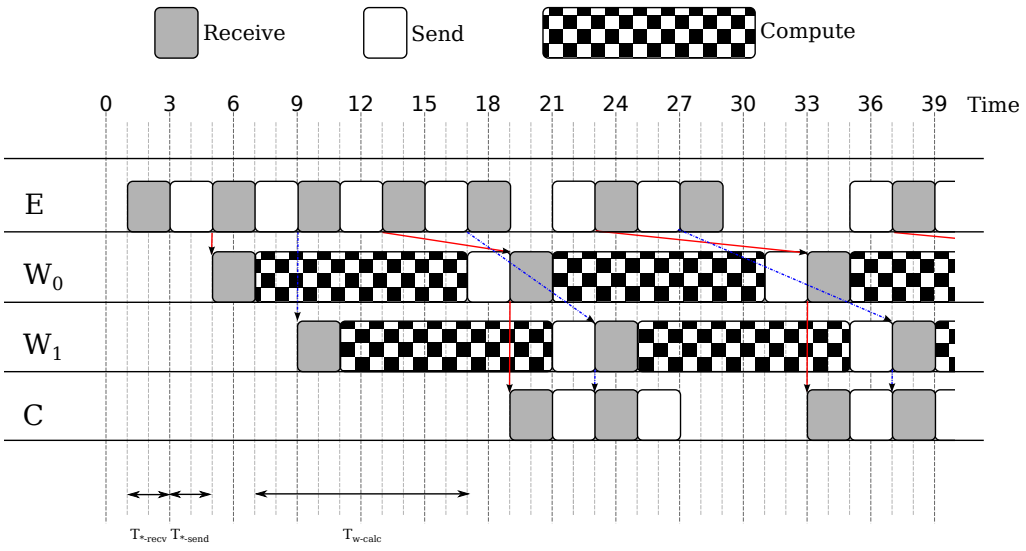


Figure 4.9: The temporal behavior of the farm implementation with a stream, the workers are limiting the throughput of the system.

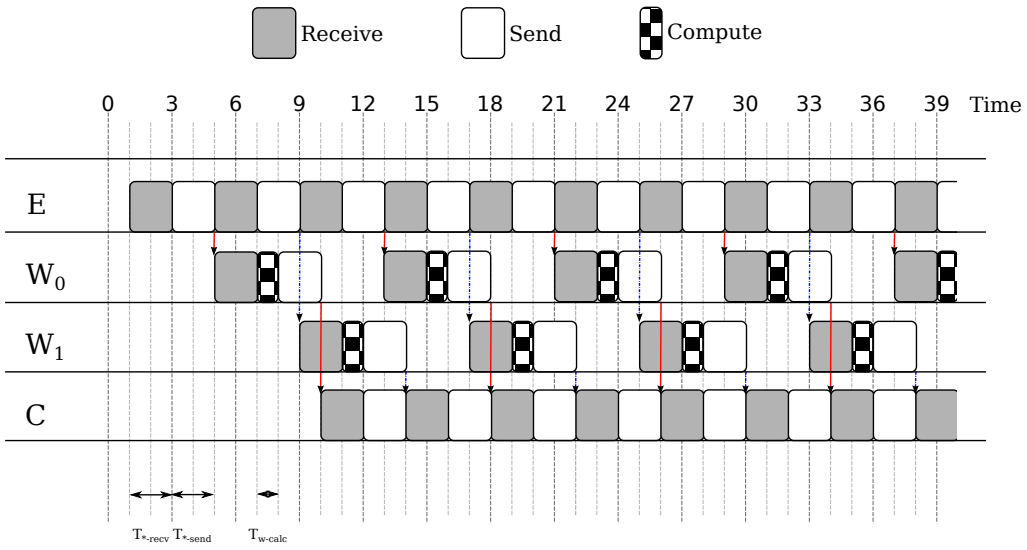


Figure 4.10: The temporal behavior of the farm implementation with a stream, the emitter is limiting the throughput of the system.

Workers are limited by the throughput of the Emitter, that is unable to sustain the number of worker. This characteristic have an enormous impact on the *service time* of the pattern.

In the general case, given the flow of tasks, we can model this implementation as a three-stage pipeline (Fig. 4.11): a task, after being served by the Emitter, will step through one of the Workers, and then to the Collector. The *service time* of a pipeline is well known, and will not be studied here. The interested reader can refer to [139, 140] for a detailed analysis. The result is given in Eqn. 4.4: it is simply the maximum of the service time of each stage. For the sake of simplicity and completeness we also report the *throughput* (also referred with *Bandwidth*, B), i.e. the amount of task processed per unit of time, in Eqn. 4.5.

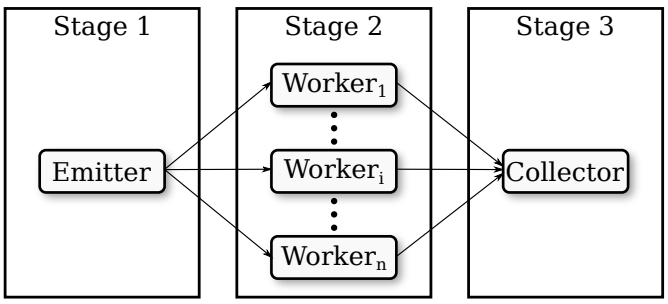


Figure 4.11: Stream-oriented pipeline modeling of the implementation.

$$T_{S-FARM} = \max\{T_{S-E}, T_{S-W}, T_{S-C}\} \tag{4.4}$$

$$B_{FARM} = \frac{1}{T_{S-FARM}} = \min\{B_E, B_W, B_C\} \tag{4.5}$$

Now, given the fact that the Emitter and the Collector are sequential entities, they compute a task every $L_{\langle E,C\rangle}$, so we have that their service time match the latency:

$$\begin{aligned} B_E = \frac{1}{T_{S-E}} &= \frac{1}{T_{E-recv} + T_{E-send}} \\ B_C = \frac{1}{T_{S-C}} &= \frac{1}{T_{C-recv} + T_{C-send}} \end{aligned} \tag{4.6}$$

Things are quite different for the workers, because we have a set of n workers (2 in the previous examples) that works concurrently. Each one has its throughput (that can be derived as with the emitter and collector by considering the reciprocal of the latency); the cumulative throughput can be easily obtained by summing the throughput of each worker. Given the fact that the workers are all equals, their throughput is simply calculated by multiplying the single throughput by n (Eqn. 4.7). The service time is therefore the single service time divided by n (Eqn. 4.8).

$$B_W = \frac{1}{T_{S-W_1}} + ... + \frac{1}{T_{S-W_n}} = \frac{n}{T_{W-recv} + T_{W-calc} + T_{W-send}} \tag{4.7}$$

$$T_{S-W} = \frac{1}{B_W} = \frac{T_{W-recv} + T_{W-calc} + T_{W-send}}{n} \tag{4.8}$$

4.2.5 Evaluating the model parameters

We reached a set of equations to model the *service time* and the *latency* of our task-farm pattern implementation. Still, we have a set of parameters that need to be estimated, using data collected from the deployment architecture and from the algorithm. Nevertheless, we confined the algorithm code to a single parameter, T_{W-calc}, while the others are just communication latencies that do not depend on the programmer code.

In general, to evaluate these parameters we can use any of the three main techniques of performance evaluation. Given the **analytical modeling** of latency and service time, we would prefer the same formalism here. However, where not possible, we can also choose to estimate the parameters by means of **measurements**, given the presence of both the architecture and the parallel program code (created by our compiler).

Shared resources in parallel architecture

By evaluating the parameters, we want to take into account the fact that multiple processes are executed *concurrently* on the parallel machine. This means that an isolated study of the behavior of each process will probably produce different results w.r.t the concurrent execution because of *interference* between processes. Nevertheless, an isolated study is much simpler, both from a modeling and from an evaluation point of view, than a global study.

For this reason we selected a mixed approach: starting from the isolated study, we count the possible interference, so that we are able to derive the global behavior with sufficient precision. The main source of interference is the presence of *shared resources*: each processor (and therefore process) compete for its access and, of course, this is not captured by an isolated study where the whole system is reserved for a single process. Current multi-core architectures provide a wide set of shared resources, such as:

a) **cores**, because of multithreaded implementations (such as SMT[168] or its commercial name *Hyper Threading*);

b) **caches**, because of (multiple) cache levels shared among groups of cores (especially the LLC usually shared among all the cores);

c) **memory**, to allow sharing of data between processes in the parallel architecture.

Of course, although possible, an interference model able to capture all the shared resources becomes very complex; for the moment, in this thesis, we consider only the most important source of interference, present by definition in all multi-core architectures: **the shared memory**. To address and evaluate the time overhead related to this kind of interference, we will define our parameters as the sum of:

1. a **fixed time**, evaluated on all those operations that do not rely on the memory (i.e. register-register operations, load/store that generates a cache hit, etc.), and therefore not influenced by the shared resources.

2. a **variable time**, evaluated as the number of memory operations that, by definition, are affected by the interference.

4.2.5.1 Evaluating the sequential time

The evaluation of T_{w-calc} is the only part that requires the knowledge of the application code. Because of isolation, we only need to estimate the execution time of the sequential source code in the target architecture. This is surely an *easier* problem w.r.t. evaluating the performance of a parallel application. Several works aimed to solve this problem exists in literature, so that we can safely consider this a solved, or at least solvable, problem that do not require further studies in this thesis.

For example, we find of extreme interest the approach in [121] that exploit a mix of static and dynamic analysis to predict the performance of a sequential program *independently* of the running architecture. In particular, the methodology deeply analyze the binary code created for a specific architecture, by means of static analysis, to derive the execution paths of the application. The code is then instrumented to gather run-time information for those paths marked as interesting by the static analysis. The frequency of each execution path and the memory access pattern is then extrapolated by means of specific models, to predict the program behavior in a parametric way w.r.t data structure sizes, for an abstract execution architecture.

Finally, by instantiating the execution architecture (i.e. the amount and type of execution units of the processor, the sizes of cache levels, etc.) and the problem size, the methodology allow us to derive the miss frequency for the various cache levels and the execution time of the application. To validate the methodology, the authors modeled the behavior of some applications from the NAS[21] benchmark toolkit using a Sun UltraSPARC-II system, and successively predicted the performance of an Alpha R12000 CPU, obtaining results usually within a 10% relative error. By following this approach we are able to gather all the information required to estimate our T_{w-calc}.

Nevertheless other, simpler approaches exists in literature [94, 96, 136, 187]. For the sake of simplicity, in our thesis propose use a very simple methodology based on the actual execution of the sequential program on our target machine. In particular, we use the following execution time model:

$$\begin{aligned} T_{w-calc} = & \quad CPU\_execution\_clock\_cycles \\ & + \quad Memory\_stall\_clock\_cycles \\ \\ = & \quad CPU\_execution\_clock\_cycles && (4.9) \\ & + \quad \sum_{i=1}^{n-1} L_i\_misses \times time(L_i\_miss) && (4.10) \\ & + \quad L_n\_misses \times time(Memory\_Latency) && (4.11) \end{aligned}$$

That essentially separate the CPU time and the memory hierarchy latencies. Despite its simplicity, this is actually used in many performance evaluation works (such as [94]) and is believed to model the behavior of a program with sufficient precision. In our case we split the time in memory accesses and cache hits, assuming an entirely private cache hierarchy, so that the only shared point in the architecture, and therefore the part that affect the variable time of the model, is the memory level.

We have that the first two addends of the formula (4.9 and 4.10) compose the **fixed time** of our T_{w-calc}, while the third (4.11) represents the **variable time**, that will depends on the amount of total requests sent to the shared resource (i.e. the memory):

$$T_{w-calc} = Fixed\_Time + L_n\_misses \times time(Predicted\_Memory\_Latency) \quad (4.12)$$

By fixing the problem size and the execution architecture, our approach does not need specific instrumentation to gather these values from the source code, but can only rely on the measurement of some hardware counters during the execution.

Current processor, unfortunately, do not count the exact parameter $Fixed\_Time$ we need for our approximation. In particular, we are usually able to measure the total execution time and the so-called *load/write stall time*, i.e. the time a processor wait for data from the whole cache hierarchy (4.10 + 4.11), but not the load/write stalls that results in memory accesses (4.11). We can, however, measure the number of cache miss for each cache level, and evaluate the sequential memory latency by using specific benchmarks.

For example, for the processors used in this thesis, we can use the performance events in Tables 4.1, 4.2 and 4.3 to gather the needed values, and then derive the two parameters as in Table 4.4

It is important to notice that in Table 4.4 we use two different values for the memory latency:

i) **Measured_Memory_Latency**, that is the memory latency in a sequential program that issue only one memory request at a time, *measured* by means of a benchmark previously executed on the target architecture;

ii) **Predicted_Memory_Latency**, that is the memory latency in a parallel program, with multiple cores issuing memory requests concurrently, *predicted* by means of specific architecture-based models that will be introduced in the rest of the chapter.

4.2.5.2 Modeling communications latencies

For the communication latencies we can derive a similar model, by analyzing the implementation of the communication primitives *send* and *receive*. For this example, we will study the implementation in Listings 4.4 and 4.5, that represent a channel implemented by using a `FIFO` queue. For the sake of readability we also list the structure of the queue (Listing 4.6), that contains a circular buffer used to store elements, a mutex and two condition variables to handle critical section and blocking in cases of full/empty channel. We highlight some important points of the implementation:

- The semantics of mutexes and condition variables follows the *posix* specification: for example when we perform a `cond_wait` we atomically enter in the condition variable waiting queue and release the mutex; when the process is woken up, it automatically reacquire the mutex, so that at the end of the execution of `cond_wait` we already are the only entity working within the critical section.

- To increase non-determinism, the waiting on the channel is not implemented with a *FIFO* behavior: when the send put a message, for example, it does wake *all* the waiting receivers. Then, all the receivers will try to acquire the mutex

Parameter	Event(s)
T_{w-calc}	CPU_CLK_UNHALTED
L_n_misses	LLC_MISSES

Table 4.1: Performance events used on Intel processors.

Parameter	Event(s)
T_{w-calc}	CPU_CLK_UNHALTED
L_n_misses	L3_CACHE_MISSES

Table 4.2: Performance events used on the AMD processor.

Parameter	Event(s)
T_{w-calc}	ONE
L_n_misses	LOCAL_DRD_MISS + REMOTE_DRD_MISS + + LOCAL_WR_MISS + REMOTE_WR_MISS

Table 4.3: Performance events used on the Tilera processor.

Parameter	Model
$Fixed\_Time_w$	$T_{w-calc} - (L_n\_misses \times Measured\_Memory\_Latency)$
$Variable\_Time_w$	$L_n\_misses \times Predicted\_Memory\_Latency$

Table 4.4: Modeling of fixed and variable time starting from evaluated parameters.

```
send(message msg, channel ch){
  lock(ch.mutex);
  while(ch.elem == ch.size){
    // Wait for a free space
    cond_wait(ch.full,ch.mutex);
  }
  if (ch.elem == 0){
    // Wake up all the waiting
        receivers
    cond_broadcast(ch.empty);
  }

  // Send the message
  ch.tail = (ch.tail+1)
            % ch.size;
  ch.len[ch.tail] = msg.len;
  memcpy(ch.buffer[ch.tail],
         msg.value,msg.len);
  ch.elem++;
  unlock(ch.mutex);
}
```

Listing 4.4: Implementation of *send*.

```
receive(message msg, channel ch){
  lock(ch.mutex);
  while(ch.elem == 0){
    // Wait for an element
    cond_wait(ch.full,ch.mutex);
  }
  if (ch.elem == ch.size){
    // Wake up all the waiting
        senders
    cond_broadcast(ch.empty);
  }
  // Receive a message
  msg.len = ch.len[ch.head];
  msg.value = malloc(msg.len);
  memcpy(msg.value,
      ch.buffer[ch.head],msg.len);
  ch.elem--;
  ch.head = (ch.head+1)
            % ch.size;
  unlock(ch.mutex);
}
```

Listing 4.5: Implementation of *receive*.

```
typedef struct channel {
  mutex_t mutex;
  cond_t empty;
  cond_t full;

  int size;
  int head;
  int tail;
  int elements;
  void **buffer;
  int * len;
} channel_t;
```

Listing 4.6: Fields of the structure *channel*.

again and, of course, only one of them will be able to consume the inserted message; the other will find the channel empty and return in a waiting state.

- Messages are considered a simple array of bytes; more complex structures will need a serialization/deserialization phase that will be performed (in our farm example) inside the worker - and therefore included in T_{w-calc}.

- Messages *can* be of variable size, so we also need to store the length of each message. The buffers inside the queue can be sized by using an upper limitation to message sizes. For the same reasons inside the receive each message structure is allocated by using the exact size of the received message.

Now that we have an implementation, we easily model T_{*-send} and T_{*-recv} by following our modeling approach:

$$T_{*-send} = Fixed\_Time_s + Variable\_Time_s \tag{4.13}$$

$$T_{*-recv} = Fixed\_Time_r + Variable\_Time_r \tag{4.14}$$

As previously explained, for our model we need to measure the time needed to perform a send or a receive in a block-free execution, i.e. when the communication does not block because of a full or empty channel, respectively. Therefore, we basically have a limited number of simple operations on the data structure that represent the channel *plus* a copy of the message to/from the channel. This code consist mainly in load/store, so we have a negligible computational time with an high number of memory transfer. We can therefore model T_{*-send} and T_{*-recv}, with reasonable precision, with a $Fixed\_Time_{\{r,s\}} = 0$. For the variable part, we need to know how much data we transfer, i.e. the length of the messages that, however, is not always fixed because of the possible variable message length. We have two options here:

a) **Worst Case analysis**: we consider the upper limitation on the size of the messages (also required to correctly size the buffers of the channel);

b) **Average Case analysis**: we consider the average length of messages;

As we are more interested in the average performance of the program, we usually prefer the second option.

Given the size of the messages, we have that a good approximation for our parameter is:

$$T_{*-send} = 0 + \frac{Message\_Size}{L_n\_Block\_Size} * Predicted\_Memory\_Latency \tag{4.15}$$

$$T_{*-recv} = 0 + \frac{Message\_Size}{L_n\_Block\_Size} * Predicted\_Memory\_Latency \tag{4.16}$$

In this case the cost of send and receive is the same, as both execute a copy of the message. There are, however, more interesting (and complex) implementations that can limit the number of total copies to one; in this case we will have a different cost (i.e. a T_{*-send} like 4.15 and a $T_{*-recv} = 0$). For this example we deliberately used a straightforward implementation for the sake of simplicity.

4.2.6 The final model for the task-farm example

We briefly summarize here the result of the steps from section 4.2.2 to 4.2.5: a model for estimating the latency and the service time of the studied task-farm implementation.

The resulting model is composed of measured values (i.e. $Fixed\_Time_w$ and L_n_misses) obtained through the execution of isolated sequential parts of the application, values modeled by manually analyzing the pattern-related code (i.e. $\frac{Message\_Size}{L_n\_Block\_Size}$) and a further parameter, $Predicted\_Memory\_Latency$, to model the interference generated by the use of a shared memory. The latter is an important point of our modeling, that allows us to predict the performance of the system by using small, isolated measurements, and then introduce a **corrective factor** that will take into account the presence of shared resources that affects the overall performance of the parallel application. The estimation of this corrective factor requires a detailed model of the target architecture, that will be discussed in Section 4.3.

In Table 4.5 we summarize the estimation of the fixed and variable parts of T_{w-calc}, starting with the measured events on an target architecture (we used the event names of the Intel architecture for conciseness), while in Equations 4.17-4.24 the service time and latency of each entity and of the whole parallel pattern are presented.

Parameter	Event / Model
T_{w-calc}	`CPU_CLK_UNHALTED`
L_n_misses	`LLC_MISSES`
$Fixed\_Time_w$	$T_{w-calc} - (L_n\_misses \times Measured\_Memory\_Latency)$
$Variable\_Time_w$	$L_n\_misses \times Predicted\_Memory\_Latency$

Table 4.5: Modeling of fixed and variable time on an example architecture.

$$L_E = 2 * \frac{Message\_Size}{L_n\_Block\_Size} * Predicted\_Memory\_Latency \tag{4.17}$$

$$L_C = 2 * \frac{Message\_Size}{L_n\_Block\_Size} * Predicted\_Memory\_Latency \tag{4.18}$$

$$\begin{aligned} L_W = {} & Fixed\_Time_w \\ & + L_n\_misses * Predicted\_Memory\_Latency \\ & + 2 * \frac{Message\_Size}{L_n\_Block\_Size} * Predicted\_Memory\_Latency \end{aligned} \tag{4.19}$$

$$T_{S-E} = L_E \tag{4.20}$$

$$T_{S-C} = L_C \tag{4.21}$$

$$T_{S-W} = \frac{L_W}{n} \tag{4.22}$$

$$L_{FARM} = L_E + L_C + L_W \tag{4.23}$$

$$T_{S-FARM} = max\left(T_{S-E}, T_{S-C}, T_{S-W}\right) \tag{4.24}$$

4.3 Performance degradation on shared memory architectures

With this section we finalize our performance modeling methodology by introducing the mechanisms used to predict the behavior of shared architectural resources in the parallel execution. In particular, our focus is on ***Predicted_Memory_Latency***, that represents the only remaining parameter of the model. For this purpose we rely on many works that emerged when shared memory architectures became popular. The first is probably dated back to 1975, when Bhandarkar [36] introduced the main idea of modeling the Processor-Memory subsystem of a shared memory architecture, considering the memories as servants and the processors as clients. The concept is that the behavior of a processor in the system can be exemplified as follows:

i) Processor P_i execute some "internal work", i.e. decoding instruction, executing integer/floating operations, etc.; in this step P_i works independently, without

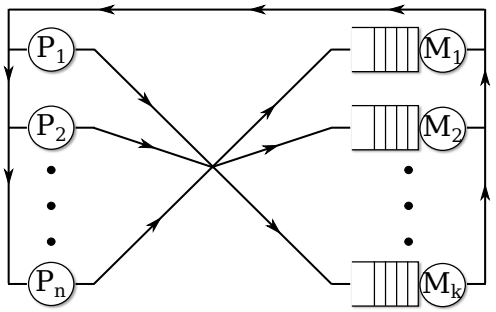


Figure 4.12: Bhandarkar's queueing network to model the memory contention.

the need of the memory;

ii) Processor P_i execute a memory read/write request, that is issued to the destination memory; P_i is now *stalled*, waiting for the result from the memory to continue the execution;

iii) The request travels through the interconnection network and reaches the memory; it is enqueued on the corresponding memory controller;

iv) When its turn is reached, the read/write operation is handled by the memory and a result is produced;

v) The results is sent back in the network and reaches the issuing processor P_i that completes the operation and return executing Point i.

This loop is executed by each processor, for the whole execution; its behavior can be easily modeled by the queueing network in Fig. 4.12: we use a closed network because the number of tasks in the system is fixed and it depends on the number of processors. In this initial definition each processor can emit a single memory request at a time; we can notice in Fig. 4.12 that the processors do not need a queue, as they always receive a single task at a time. Nevertheless, they behave as servers because, when a task is received (i.e. a memory response) they execute Point i for a certain time, before sending the task again to the memory (representing a new request, Point ii). Each memory can be straightforwardly modeled as a FIFO queue followed by a server.

Parameters to instantiate the queueing network and derive performance indexes can be captured by determining the proper probability distribution for the duration of each of the previous point (e.g. point i can be characterized by using an exponential distribution with average t_p). Another important point is that the memory access pattern determines the distribution of requests among the different memories; in [36] the author modeled the problem as a sequence of Bernoulli trials. The

modeling effort is not further discussed in the paper, mainly focused on resolution methods for this queueing model.

This queueing network allow us to estimate ***Predicted_Memory_Latency***, by deriving the **Response Time** of the memory servers, that consist in the *Service Time* of the memory *plus* the average *Waiting Time*, i.e. the time spent by a request waiting for its turn. The model can be easily parameterized by using exponential distributions, that represents a good approximation.

The processor server will use a service time calculated by considering its total working time $Fixed\_Time_w$ and the number of memory requests executed, assuming that memory requests are uniformly distributed among the whole execution time:

$$T_P = \frac{Fixed\_Time_w}{L_n\_misses}$$

The memory server, on the other hand, will use $T_M = Measured\_Memory\_Latency$ as its service time.

4.3.1 Extensions to the original queueing network

Research on Bhandarkar's seminal work followed two main branches:

- the study of computationally simple model approximations, to solve the queueing network efficiently;
- the extension of the model to match the evolution of computer architectures.

Indeed, although simple from a conceptual point of view, this model was already too complex to be solved by using the then-state-of-the-art results in queueing network. In our case, we are not (yet) interested in efficient methods to solve the queueing network, so we are not giving more details on this specific topic.

Focusing on the second point, this simple yet elegant model has been extended in the following years, to cover the increasing complexity in parallel architectures.

4.3.1.1 Modeling caches

Among the most important works, we cite again Bhandarkar[36] that, in the last part of the paper, proposed an extension to deal with caches, based on the simple concept of *cache hit probability* (p_h): among the whole set of memory request T, a fraction will be completely handled by the cache ($T * p_h$), and only the rest will reach the memory. A queueing network to handle caches is depicted in Fig. 4.13: each memory request is sent to a server, that represent the cache; with a certain time t_c the cache will finish its work and will either:

1. produce an *hit* with $p = p_h$: the task is sent back to the processor;

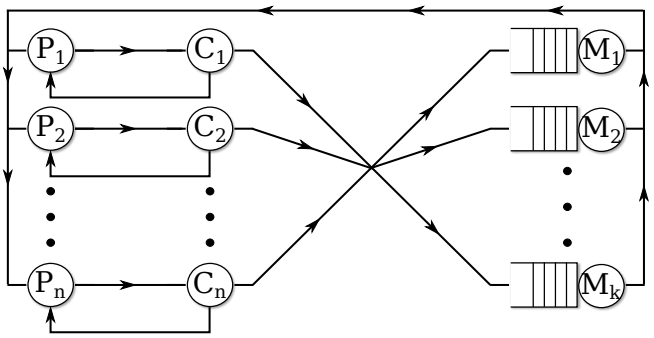


Figure 4.13: Queueing network model for systems with caches.

2. produce a *miss* with $p = 1 - p_h$: the task is sent to the memory, as in the simple case;

This behavior, of course, can be generalized to model an entire private cache hierarchy.

4.3.1.2 Bus interconnections

In the following years the model was extended to cover the widely used bus interconnection; the queueing network for multiple-bus systems (a widely adopted solution for parallel machines at the time) is depicted in Fig. 4.14, following the solution described in [37]. The bus(es) itself is represented by a queue that process tasks (requests from the processors or replies from the memories) once at a time. Several analytical solution based on this or very similar queueing networks have been studied [97, 122, 164], proving the effectiveness and correctness of this model.

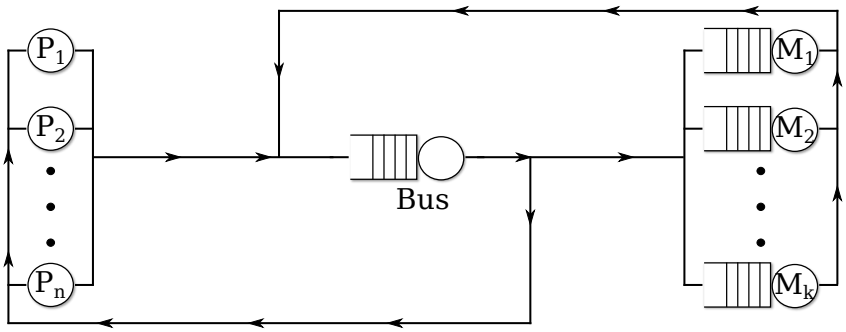


Figure 4.14: Queueing network model for multiple-bus interconnections.

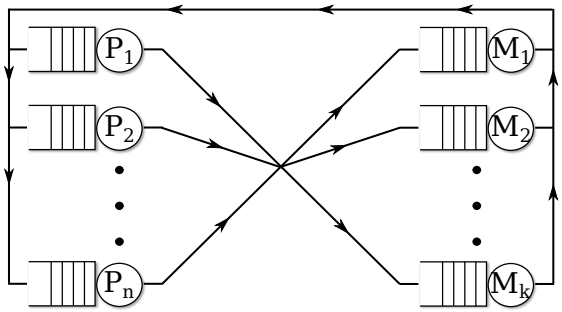


Figure 4.15: QN model for systems with multiple requests per processors.

4.3.1.3 Multiple Requests per processor

With the emergence of superscalar and out-of-order processors, the assumption of having a single task per processor in the system became too restrictive to effectively model this kind of systems. From the queueing network point of view, however, a straightforward update of the network is possible, by adding a queue to the P servers and fixing the number of tasks per processor to a value higher than 1 (depending on the architecture to be modeled). By transforming P in a service center (Fig. 4.15), we are able to express the possibility of generating multiple requests, up to the amount of tasks in its queue. When all tasks in the queue are consumed, P has reached its maximum number of outstanding requests and blocks. At each response, it will restart the computing phase to produce a new request.

With this modeling, it is extremely important to select the proper amount of requests per processors, that *depend on the executed code*. Indeed we can use the hardware limit (i.e. the maximum amount of outstanding requests), but this will probably represent an higher number than the real amount of concurrent requests issued by the processor because of data dependencies that cannot be avoided even by using out-of-order processing. Therefore it is important, if we want to analyze the performance of a specific program, to derive in some way this parameter from the code. This extension has been used, for example, in [183].

4.3.1.4 Complex interconnection networks

Crossbars and buses represents two opposite corners in the world of interconnection networks: the first allows the maximum number of independent connections concurrently at the cost of being the most expensive interconnection; the latter, on the other hand, allows a single communication at a time, but is also very cheap. Many interconnections were proposed in the middle, to increase concurrency with limited costs. In the years, the group of so-called multistage networks (that includes, among the others, banias and butterfly networks) gained a lot of popularity. The generalization of the model to multistage networks is indeed quite difficult and, at

the moment, a limited number of works try to solve the problem. Of particular interest is the modeling effort of Willick[183], that extend the base QN model by considering a service center for each output port of each switch, and class-based routing to correctly "forward" tasks from processors to memories and back. The work also consider multiple-packet messages, to better handle the packet-switching nature of this class of interconnection networks.

Starting from this queueing network, an analytical approximation is derived by using approximate mean value analysis. The model is validated by comparing analytical results with a simulation of the behavior of a real multistage network, resulting in extremely low errors with architectures up to 128 processors and memory interfaces.

4.3.1.5 Cache coherency

Another sensible problem of current architectures is the presence of cache coherence protocols, that massively affect the data exchange pattern inside a parallel architecture. In hardware-based cache coherency, caches exchange data to synchronize themselves on write operations. The most common protocols provide invalidation mechanisms, to remove obsolete copies of cache blocks. This increase the cache miss rate and, depending on the interconnection networks, requires the use of invalidation messages. Modeling the behavior of cache coherency analytically is still an open research, because of the strong implications of the program (in particular its data access pattern) on the coherence traffic. The problem was addressed by adopting several simplifications on the workload model, and analytically deriving the coherence overhead by using tools such as Markov chains [73] or Generalized Timed Petri Nets [176]. Among the others, the work of Yang et all [188] is quite interesting as it uses a queueing network that indeed represents an extension of the original model of Bhandarkar. The resulting network is quite complex, as it mixes both closed and open classes of customers, the latter needed to model the cache invalidation traffic. The resulting QN is depicted in Fig. 4.16.

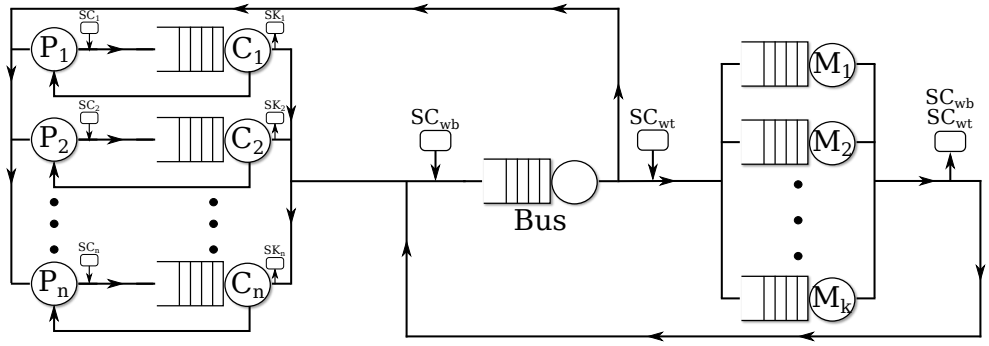


Figure 4.16: Queueing network model for cache coherent systems.

4.3.2 Adapting the model to a concrete parallel architecture

We now present an application of this model to the Symmetry S-81 parallel architecture, studied by Tsuei and Vernon[165]. This represents a significant application of our approach, able to measure several metrics, including the **Response Time** of the memory servers. The resulting Queueing Network is depicted in Fig. 4.17; without entering in specific details, one of the main characteristics of this model is that uses different queues to address specific operations; all those queues, of course, competes for acquiring the bus. In particular we have:

- **iv**, that represents a cache invalidation request;
- **r**, that represents a read requests, that will be addressed by the shared memory **or** another cache;
- **rw**, that represents a write requests, that will be addressed by the shared memory;
- **memrp**, that represents a shared memory response;
- **cacherp**, that represents a remote cache response.

This model has been validated against the real architecture by using two parallel programs, resulting in a measured error always within 9%. This proves the accuracy of the model and the feasibility of queueing-network models to analytically predict the behavior of a system.

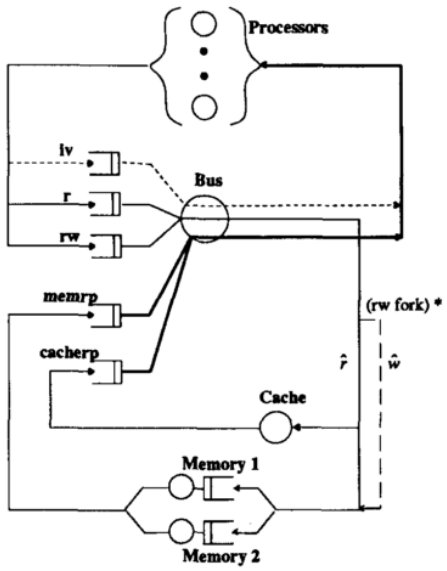


Figure 4.17: Queueing network model for the Symmetry S-81.

4.4 Summary

In the previous chapter we discussed the need of a performance model, to allow the parallel compiler to compare the possible implementations and determine the best (or at least a good) solution. In this chapter we addressed the problem by using queueing network concepts; the result of this chapter not a well defined model, but a set of tools and modeling building blocks that can be used to create the required model.

We decomposed the modeling problem in three points:

1. Evaluate the performance of the application as a graph of parallel modules;

2. Evaluate the performance of each parallel module;

3. Evaluate the performance degradation of shared memory.

Because of the strong implications of the parallel pattern and the target architecture on each point, we are not able to offer a general queueing network for these points: the model have to be studied on a per-pattern and per-architecture basis.

Nevertheless, we were able to adapt an already existing methodology for Point 1, that is general enough to abstract from these characteristics and allow us the study of stream-based parallel applications; a similar approach is proposed for entirely data-parallel applications composed of several modules; this, however, needs further refinements (that are left as possible future works) to allow its applicability in general cases.

On the other hand, a pattern-based model is required for Point 2, while a detailed architectural model is used in Point 3. These two need, of course, a specific definition for each pattern implementation and target architecture. Throughout the chapter we discussed several modeling solutions for common problems, to give an idea of the strength of QN-based models and to prove the feasibility of our approach.

In particular, for Point 2 we exploit the knowledge given by the pattern to derive the behavior in terms of communications and synchronizations; this, together with the pattern implementation details, allow us to predict the time spent by our parallel module in communications and synchronizations. The remaining time, spent by executing the algorithmic code, is evaluated by profiling the sequential code. All the times consist in a *Fixed Time*, that is not affected by the parallel execution, and a *Variable Time* that, on the other hand, depends on the presence of multiple, cooperating processors in the architecture.

The *Variable Time* is analyzed in Point 3, by taking into account of the specific details of the running architecture, and its shared resources. Here we use the well-established approach of modeling the Processor-Memory subsystem as a queueing network. Nevertheless, the different characteristics of each architecture requires the definition of a specific QN model for the target parallel machine.

In the next chapters, after a deeper analysis of current parallel architectures and parallel pattern implementations, we will be able to discuss the proper QN-based models on specific examples.

Chapter 5

A Queueing Network Model for Tilera TILEPro64™

In this chapter we study a commercially available multi-core architecture, in order to develop a detailed Queueing Network model of the chip. This model, defined by using the modeling basis provided in Chapter 4, Section 4.3, will allow us to evaluate the performance effects of the shared memory contention. From the set of architectures presented at the end of Chapter 3 we selected the Tilera Tile*Pro*64™ processor, mainly for two important points:

1. Despite its low peak performance in terms of instruction per second, it represent a very advanced multi-core chip, as it contains a very large number of cores[1], a complex interconnection network, multiple memory controllers and several interesting solutions that are likely to be integrated in other architectures in the future.

2. The architecture is extremely documented, so that we are able to understand low-level characteristics required for the development of our performance model; furthermore, the development environment offer a cycle-accurate simulator that allow us to accurately study the behavior of the chip and the performance effect of the various design choices.

The design of a Queueing Network model for an architecture can be a tricky task, that involves watching the real behavior, thinking of modeling ideas to capture the behavior, and finally test if the designed model really works as the real machine. To efficiently check our ideas we created our own queueing network simulator, **EQNSim**, that will be presented in the next section.

In the rest of the chapter, after a first overview of the architecture, we will study each key module of the chip, its characteristics and how to model it in our QN-based

[1] As of today, the only commercially available chip that contain an higher number of general-purpose cores is the Tilera TILE-Gx8072[163], that represents the direct evolution of the Tile*Pro*64, and as such shares most of its characteristics

model. At the end the overall Queueing Network Model will be presented, along with some experiments w.r.t the real architecture to demonstrate the effectiveness of the model.

5.1 EQNSim: a testing environment for queueing network models

Designing a Queueing Network model for an architecture involves several tasks:

1. Analyze the behavior of the real architecture, usually by means of specific tests or benchmarks targeted at studying specific aspects (i.e. the network latency, the memory response time, the behavior of cache coherency protocols, etc.).

2. Use the tools available to create or adapt a model that should behave, with a sufficient level of approximation, as the analyzed system.

3. Instantiate the model and verify if the required level of approximation is met.

During the definition of a model we usually try to study one effect at a time, modeling each one individually, and then mix them up to obtain a model of the overall system. This means that, by definition, these three steps will be executed several times, depending on the effects to be modeled.

Furthermore, modeling idea from point 2 may not meet our expectation, resulting in approximations not as effective as thought when verified at point 3. In this case, we need to refine the model, by changing some parts or sometimes discard it completely and restart from scratch.

In this incremental and iterative scenario we wanted a tool to rapidly prototyping our model ideas and compare them w.r.t the real architecture. In literature, queueing networks are usually solved by means of two very different tools:

- **Analytic Solvers**, such as the qnetworks Toolbox[123] and PDQ[2], that are able to analytically compute the steady-state statistics using mean value analysis, markov chains or similar approaches; the problem of these tools is that analytic solutions are known only for a (small) subset of queueing networks models (for example, a product form (required by most solvers) is only available for a very small subset of closed networks, called BCMP networks [25]).

- **Simulators**, such as Java Modelling Tools[29], able to simulate generic networks until a steady-state system is reached to gather the statistics; in this case there is no need of any kind of transformations.

[2]Pretty Damn Quick, `http://www.perfdynamics.com`

The first class of tools is actually very fast in finding the steady-state statistics; however, if the original model does not fit the required restriction, a further approximation effort is required. Given the uncertainty of the model we are solving, we did not want to spend a lot of time in adapting our ideas to the kind of networks allowed by these tools and decided to use simulators. Of course, when the complete architectural model is found, it is possible (and recommended) to study analytic solving methods: it is not convenient to equip our parallel compiler with a queueing network simulator, that could take a lot of time to reach the steady-state solution; however, in this first step towards the parallel programming environment, we are just interested in finding a good model, not on efficient ways to solve it.

Unfortunately, also most queueing network simulators are difficult to use in an environment in which the QN is to be refined continuously, where you need to change hundreds of service times per test, preferably in an automatic way. We also wanted something more powerful and customizable: **EQNSim** was born. While the name stands for **Extensible Queueing Network Simulator**, the idea behind **EQNSim** was not really to produce a queueing network simulator, but more like a "playground", where queueing network concepts could be applied, generalized and mixed with other, more architecture-related, behaviors.

Let us say, for example, that we want to model the behavior of a specific component of the architecture, such as the memory controllers or the interconnection network, and we have a modeling idea that we want to test. We could use the cycle-accurate simulator of the Tile*Pro*64, but the closed nature of the system makes the performance of each component strictly related to the behavior of the others that, however, we do not know (yet) how to model. In practice, we do not really know how to test our idea because we do not have the entire queueing network model.

A solution (or at least a starting point towards the solution) is to "mix" queueing networks *and* real architecture simulators. For example, if we are testing the memory, we could create a closed queueing network, such that proposed by Bhandarkar[36], and substitute the memory queue with a cycle-accurate memory simulator. This "model" gives us the behavior of the memory when requests are generated by following a very specific frequency pattern, making it comparable with an entire queueing network model in which we introduce our modeling idea for the memory controller.

To reach this level of extensibility we selected the well known simulation framework **OMNeT++**[174]. OMNeT++ was born mainly for research in computer networks, as a substitute for NS-2[22] and OPNET[55], but with extensibility in mind: according to the authors

> OMNeT++ represents a framework approach. Instead of directly providing simulation components for computer networks, queuing networks or other domains, it provides the basic machinery and tools to write such simulations. Specific application areas are supported by various simulation models and frameworks such as the Mobility Framework

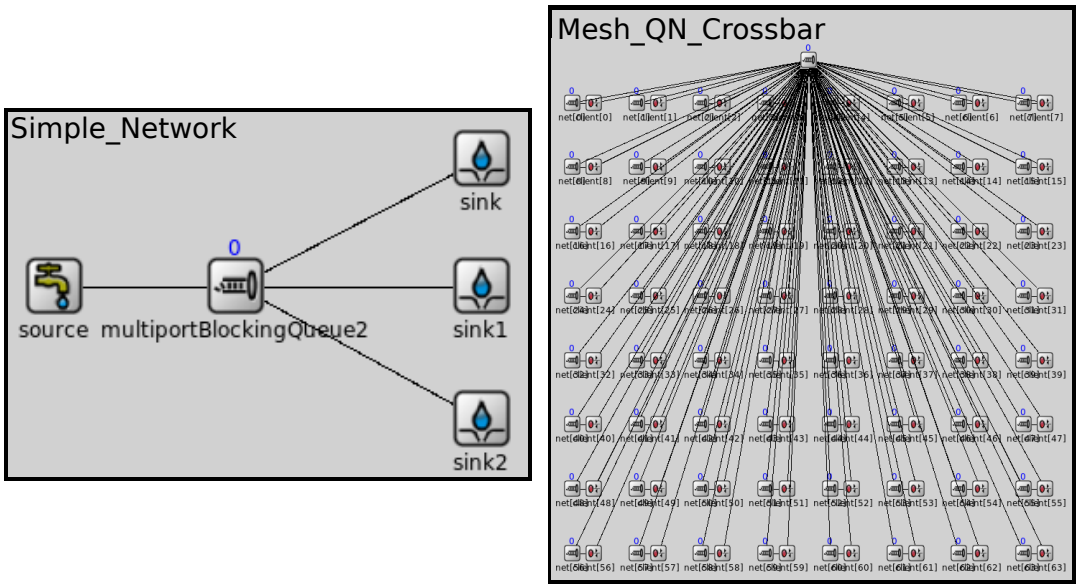


Figure 5.1: Two queueing network models represented on EQNSim.

or the INET Framework.

In practice, OMNeT++ is just a framework that provides an event-based simulator, on top of which the user can define proper entities that cooperate throughout the simulation. In the case of **EQNSim**, we developed the classic entities of queueing networks: sink, sources and queues. These components have a well defined interface, so that we can actually substitute each of them with different, more complex components, to simulate single parts of the real system, as long as they respect the interface.

To ensure the correctness of the queueing network components of **EQNSim**, we tested them by comparing the results of some networks w.r.t the same networks simulated with JMT. **EQNSim** has been extensively used throughout the thesis: the results of all the queueing networks reported in the thesis were obtained by using our tool.

5.2 Architecture overview of Tilera TILEPro64™

In this section we give an overview of the Tile*Pro*64 [26, 162], introducing the key points of the architecture. Then, in the following sections we will analyze, piece by piece, each part of the architecture to define a way to model it inside our Queueing Network Model. The starting point is, of course, the initial model by Bhandarkar[36](Fig. 5.2), that will be "extended" by using some of the already known solutions listed in the previous chapter, mixed with new ideas for the specific architecture.

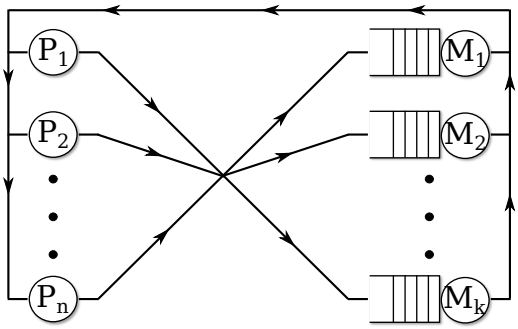


Figure 5.2: Bhandarkar's queueing network to model the memory contention.

The Tile*Pro*64 features 64 identical processing cores (tiles), interconnected with Tilera's iMesh[180] on-chip network. Each tile consists of

a) a 3-way superscalar VLIW processor;

b) a cache subsystem, composed of two separate L1 caches for data and instructions, a unified L2 cache and a Translation Lookaside Buffer (TLB);

c) a switch that implements the iMesh interconnection network.

The iMesh [180] NoC is, in fact, composed of five independent meshes, each one carrying a different kind of traffic, such as I/O transfer, memory access and cache coherence protocol. One of them is particularly interesting because it can be directly used by the programmer to exchange small messages among processors without exploiting the shared memory system. This mechanism ensure ultra low latency communications (tens to hundreds of clock cycles) for very small message (up to $125 words$, corresponding to $500 Bytes$).

To sustain the memory bandwidth requirements of 64 cores, the Tile*Pro*64 provides four on-chip memory controllers, placed at the four edges of the mesh. This means that, in general, the memory latency for a core depends on its position in the mesh and on the memory controller selected. In practice, however, several features of the architecture, such as the cache coherence protocol and the memory striping system, can attenuate this effect.

Figure 5.3 shows the architecture diagram of the Tile*Pro*64 processor. Overall, this is a far more complex structure compared to commodity multi-core CPUs, where the number of core is limited (up to 8, at the moment being), and all the cores are, in practice, equidistant with respect to the single memory controller usually available.

The Tile*Pro*64 design includes innovative, distributed and scalable approaches to handling virtual memory and cache coherence. To ease the burden of having multiple memory controllers, virtual memory pages (64KB long) can be "striped" among the physical controllers in 8KB chunks. This way memory requests are

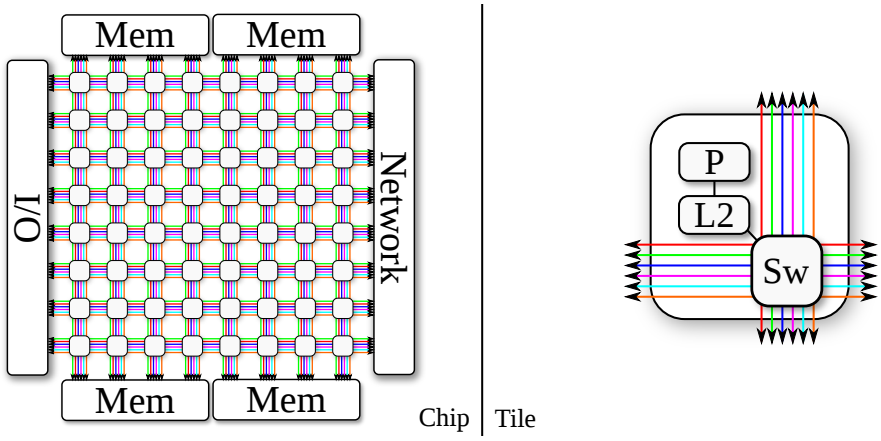


Figure 5.3: Tile*Pro*64 architecture: chip architecture (left) and single core (right).

automatically and transparently spread across the different memory controllers. Of course, if desired, the programmer may also require to directly allocate entire VM pages on particular memory controllers, as usual in NUMA architectures.

The use of a complex interconnection network have a deep impact on the cache hierarchy, and in particular the cache coherency mechanism. Indeed, a snoopy-based approach is unfeasible in such architecture. The Tile*Pro*64 implements Dynamic Distributed Cache (DDC)[160]. This is a kind of distributed directory mechanism implemented on top of L2 caches. The basic idea is that for each cache line a tile is elected to be the line *Home Node*. The Home Node is responsible for handling the coherency relative to the line, by always maintaining an updated version of the line and working as a directory, i.e. keeping the list of sharers and sending proper invalidations when needed. This "homing" mechanism is handled by using the L2 cache of the tiles, so that any local L2 space is contended by the core on the tile (for the standard cache behavior) and the others (for the DDC behavior).

The presence of a *Home Node* significantly affect the cache hierarchy: with common cache coherence mechanisms, on a cache miss, the line is usually requested to the next level of the hierarchy; a (faster) cache-to-cache transfer may occur iff another processor is using (or recently used) the same cache line. With the DDC, on a cache miss, the request is sent to the Home that, aside from working as its directory, may keep the line in its cache, even if the processor on the Home never used it. For this reason Tilera presents the DDC mechanism as a *virtual distributed L3 cache*; of course, it is virtual because implemented on top of the level two caches. The downside of this cache coherency mechanism is that, for each cache line allocated in the chip, the corresponding Home node *need* to keep an updated copy of the line. This means that, if the Home node does not overlap with one of the cores that are currently using the line, the number of copies will be $|sharers| + 1$, thus introducing a space overhead for keeping the line coherent. Given the (relatively) small amount of cache per core, the space overhead can seriously affect the performance of a

program; for this reason the Home node can be selected with several policies. The common behavior is to determine the Home by using a hash function, that ensure a uniform distribution among the 64 tiles. The programmer can modify this behavior by explicitly selecting a Home node for a slice of the virtual space. This "customized" behavior is effective when a specific sharing pattern is identified by the programmer, to minimize the space overhead of the DDC mechanism (i.e. by selecting as Home a node that is known to use the line). In addition, the Home node can be completely disabled, resulting in an incoherent mode. It is important to notice that *these DDC policies coexist inside a single process*: the programmer is able to select the best homing mechanism for each memory area. This is possible by using specific bits of the page table entries to identify, for each page, the selected policy and/or the specific Home node of the page.

The Tile*Pro*64 processor architecture defines a relaxed memory consistency model, in which both load and store can be reordered (similar to POWER architectures and in contrast to x86 architectures where store order is guaranteed). A memory fence instruction is provided to force ordering, if needed. Moreover, according to a pure RISC approach, a single atomic instruction (Test-And-Set) has been included in the instruction set. It should be noted that the atomicity is guaranteed *only* if the instruction is used on a cache coherent address.

According to a "general-purpose accelerator" philosophy, the engineers preferred to adopt a large number of general-purpose cores in place of special-purpose accelerators for specific tasks. The use of the cores is left to the programmer: if preferred, some of them can be equipped with special "driver" programs that works as intelligent interfaces w.r.t the external interfaces of the chip (i.e. networks and PCIe). Indeed it is a programmer choice to use these helper processes at the cost of reducing the number of available cores, or work directly with the interfaces at the cost of a more complex interaction.

Several degrees of freedom are also given from the operating system point of view: each tile may run its own (full) copy of the operating system, or the whole chip can be used by a single O.S. instance (in both cases a customized Linux kernel). In the latter, a specific operating mode exists (Zero Overhead Linux) to disable the operating system on specific cores (by disabling several interrupts), so that a selected process runs on the core without being interrupted by O.S. tasks (including the scheduler). Lastly, it is possible to run a program without a (properly said) operating system, by using the *Tilera Bare-Metal Environment*, that consists in a set of libraries that provides support for basic O.S. tasks (i.e. handling virtual memory and starting processes) to offer the complete control of the architecture. All the results presented in this thesis were obtained by using the Zero Overhead Linux configuration, that offered almost the same control and predictability of the Bare-Metal Environment, but with a much easier programming workflow.

5.3 Processors

The processor architecture is significantly different w.r.t the current generation of CPUs. Its design choices clearly highlight the need of a simple architecture, to save space and thus increase the number of cores per chip. Despite its simplicity, however, the design of Tile*Pro*64 cores is interesting and, in many ways, elegant. The processor clearly follows a RISC paradigm (sometimes referred as "load-store" or "register-register" architecture[92]), with operations allowed only among the 64 general purpose register, and explicit load/store instructions. The Processor uses a short, in-order pipeline that consist of five stages (Fetch, Decode, Execute0, Execute1, and Register Updating). The processor is superscalar, supporting up to three instruction concurrently (two arithmetic operations and a memory one); again, the need of simplicity pointed to the development of a VLIW instruction set[3]: each "instruction bundle" is 64-bit long and contains one instruction for each of the three execution units; the bundle must contain independent instructions (i.e. no dependencies can exist inside a bundle) and the whole bundle is stalled until all the input registers are ready. The VLIW approach is able to offer superscalar in-order processors as fast as their corresponding out-of-order version, provided that the compiler is able to correctly identify independent instructions. The size of the pipeline also reduces the amount of stalls and their impact w.r.t more complex architectures (a branch misprediction, for example, takes only 2 cycles on the Tile*Pro*64 [160] w.r.t the 14 cycles of an Intel Sandy Bridge architecture[4]).

All those features makes it possible to easily model the processor behavior and, in particular, its service time (i.e. the number of clock cycles that passes between two memory requests; the only point of interest in our model is the load/store subsystem, that allows more than one pending operation, meaning that the processor can send multiple memory requests. However, given the in-order processing, we *approximate* the system by considering a single request at a time. Thus, for the processor part, we keep the single queue of the starting model.

5.4 Cache Hierarchy and Coherency

As previously pointed, the cache subsystem of the chip is quite common (two levels of private cache), but the innovative cache coherence mechanism is likely to become a modeling nightmare. We briefly analyze the different possible working modes of the cache coherency mechanism, to finally end with our motivations to model programs in which cache coherency has been disabled.

[3]The Tile*Pro*64 is one of the few processors implementing a VLIW approach today, along with the Intel Itanium and some embedded processors.

[4]Sandy-Bridge latencies are reported at `http://www.7-cpu.com/cpu/SandyBridge.html`

5.4.1 Hash-for-Home

This represent the default approach to automatic cache coherency, where the DDC is distributed among all the caches, consuming approximately the same amount of L2 on any tile. This DDC policy is extremely important when the application uses a single core. In this case, with a common cache coherency mechanism, the single process would have access only to its local cache: any L2 miss would end in a memory request, and the other caches would remain unused. By using DDC, the local level two cache is partially used as a directory, but only for cache lines used by the core itself, resulting in no cache space loss. On the other hand, the other caches devote all their space (that would have been otherwise unused) as DDC. This way a local L2 cache miss, have still a chance of finding the cache line on its Home node, without the need of reaching the slower shared memory. This is the best use case of the DDC as a virtual L3 cache, as all the other caches will contribute as a further hierarchy level. Of course, the same behavior can be experimented when a limited number of cores is used. This policy can become ineffective when all the cores (or a large number of them) are exploited for the computation.

From the modeling point of view, with this policy each cache miss will produce a request towards the selected home, that is uniformly distributed among all the tiles. This means that, in theory, each of the cache would also become a *server* that receives requests from any core and, eventually, forward the request towards the memory. We should also consider that the *miss probability* of each cache is changed, because a large portion of the cache may be used as directory. We would have a completely different queueing network, difficult both to be parameterized and to be evaluated.

5.4.2 Single-Home

With the Single-Home policy the programmer is able to manually select the directory node for a portion of memory. This is important partially to solve the space overhead of using automatic cache coherence, as the directory can be selected to overlap with one of the cores that is actively using the data.

From the modeling point of view here we partially avoid the problems on the *miss probability* of the caches, as the space we are devolving for the directory can be effectively used also as cache for the local processor. We still retain the problem of modeling the caches as a further level of servers, and the additional hop to reach the directory node when needed.

5.4.3 No-Home

With the No-Home policy we simply disable the automatic cache coherence. The flow of a request is much simpler here: if the cache does not have the required address, a request to the memory is issued, as common in most incoherent or

snoopy-based architectures. In this case, we can use the original model with no modifications.

5.4.4 Restriction on the model

To keep the model simple (we already have many characteristics to be correctly modeled and verified) we decided to allow the parallel compiler to only use the **No-Home policy**, thus requiring to handle cache coherence at software. Of course, this does not mean that the application developer will have to explicitly handle cache coherence: this will still be a duty of the parallel compiler. However, requiring the parallel programs to not use automatic cache coherence **may** result in not fully exploiting the architecture. This is still a debated issue, that will be further addressed in later chapters. For the moment we can say that we consider this possible performance loss aligned with the concept of finding **predictable and good** (and only preferably the best) implementations of the parallel pattern to be able to compare them.

5.5 Interconnection Network

We now consider the interconnection network of the chip. The presence of five independent meshes simplify our work, because we can focus only on the one that is used for memory transfers, allowing us to avoid any other source of traffic (e.g. processor-to-processor messages, invalidations and I/O messages), that could affect the under-load response time but are handled by independent networks.

The mesh interconnection exploit wormhole switching [92]. Each message (packet) is composed of a header plus a variable number (up to 128) of flits[5] (32bytes long); the header is used by the router to (statically) determine the destination. Each switch is able to forward a flit in a single clock cycle. The use of a single, chip-wide, clock domain makes all the switch synchronized, such that a flit forwarded by a switch at cycle t is received and handled by the next switch in the path at cycle $t + 1$. This, paired with a credit-based flow-control mechanism, allow each switch to forward a flit in every cycle, as depicted in Figure 5.4. In the example an 18 flit long packet is sent from SW_1 to SW_3, through a path long 4 hops. From the Figure we can derive a simple formula to predict the forwarding time of a packet (i.e. the time between the entering of the header in the network and the arrival of the last flit to the destination) given its size (m, including the header) and the number of hops in the interconnection network (d), as depicted in Eqn. 5.1

$$(d - 1) + m \tag{5.1}$$

[5] A flit (**fl**ow control dig**it**s) represent the amount of data sent in a single communication between hardware units, and thus the smallest unit of flow control.

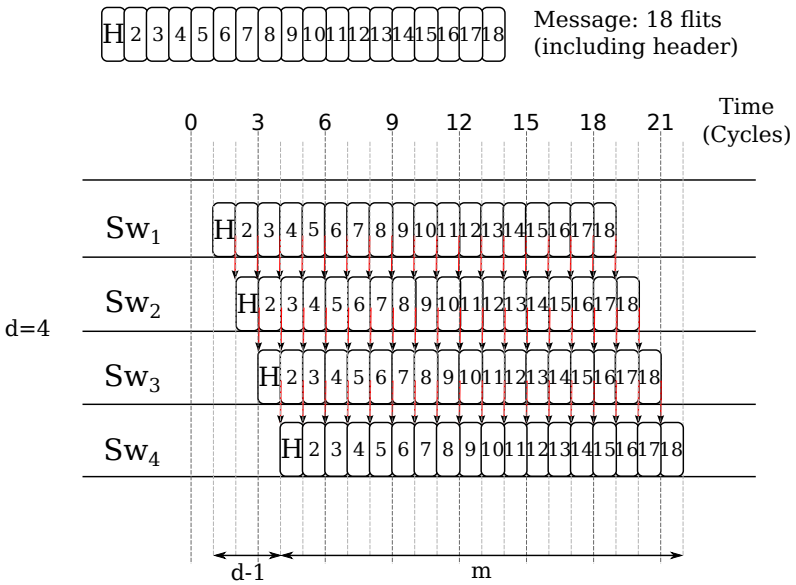


Figure 5.4: Time diagram for the dispatch of a packet in the iMesh network.

Of course, the latency may increase if the network is loaded and multiple packets compete to acquire the switch resources. The estimation of under-load latency will be addressed later.

While correct in general, Eqn. 5.1 needs to be adjusted to consider important characteristics of the iMesh interconnection.

1. iMesh switch usually takes a single cycle to forward a flit. However, according to [180], the header forwarding takes an extra cycle if the packet "must make a turn" (e.g. a packet received from the left/right interface must be forwarded on the up/down one).

2. Routing on iMesh is statically defined, and follows a dimension-ordered routing policy: giving the 2D nature of a mesh, a packet is first forwarded to match the correct row/column of the destination, then on the second dimension until the destination is reached. The Tile*Pro* family of processors allow each network to be configured to either route X first, or Y first. In particular, our experiments demonstrated that the memory network follows an XY routing, where packets are first forwarded towards the correct row, and then towards the destination column. This routing has an interesting property: packets that flows from a core to the memory (i.e. memory requests) follow a different route than packets that flows from the memory to a core (i.e. memory responses). In Figure 5.5 are shown, as example, the routes for node $\langle 5, 3 \rangle$.

3. Each memory interface is connected to multiple switches of the mesh: using a single switch would cause all the requests to be directed towards that switch

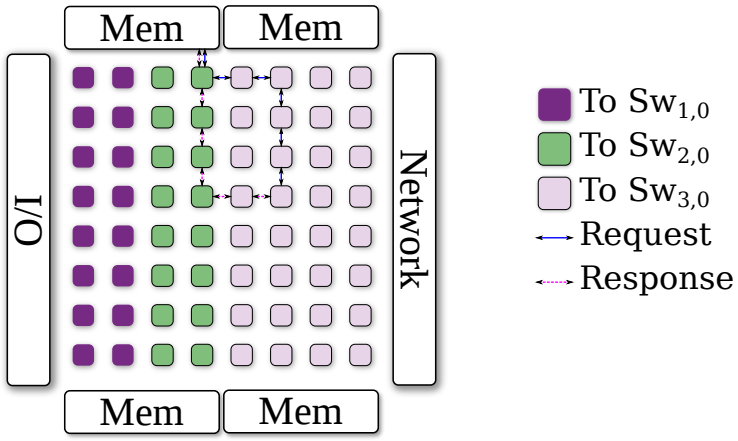


Figure 5.5: Routing paths on Tilera: destination interfaces for requests to the first memory controller and request-response paths for node $\langle 5, 3 \rangle$.

that, according to Tilera documentation[160], would become a bottleneck. By following a simple assumption on the maximum theoretical bandwidth of the memory controller, we obtain that the switch should support a bandwidth of (according to Tilera documents[160]):

$$MaxBW_{MemCtl} \simeq 52Gbps \tag{5.2}$$

while a single switch has a maximum bandwidth (considering a 32-bit flit per cycle) of

$$MaxBW_{SW} = 866 * 10^6 * 32 \simeq 27Gbps \tag{5.3}$$

So a single switch can indeed become a bottleneck. The architects selected three switches, which give a total bandwidth of $\sim 83Gbps$, enough to guarantee that this connection does not become a bottleneck even under its maximum load, assuming that the requests are equally distributed among the three switches.

Each core select its destination depending on its position (specifically, its "column") on the mesh. This is depicted in Figure 5.5 for the first memory interface (upper left). It is interesting to note that the fourth column, despite having a direct connection to the memory interface, routes its packets towards the third column, reserving the interface on the fourth for the other half of the mesh. The fact that two switches are allocated for the nearest half of the mesh, and only one for the other can still, at least in theory, produce bottlenecks (e.g. if all the memory bandwidth is exploited by the wrong half of the mesh), and may encourage the use of NUMA allocation to manually select the correct memory controller for each core. The same configuration is used for the corresponding interface on the bottom (bottom, left), while for the two right interfaces the matching is mirrored w.r.t the one in Figure 5.5.

4. We should also consider that request and response packets have different sizes; for example, on a cache miss, the cache will request a line, sending a packet composed of 3 flits (the header, one containing the operation request, and one containing the starting physical address); the response, on the other hand, will contain the entire cache line (64Bytes in the Tile*Pro*64 architecture[160]), thus requiring a packet of 18 flits (the header, the operation identifier and 16 flits for the cache line). A store request (that happen when the cache is removing a modified line because of its write-back behavior) will basically be the opposite, with a request packet of 19 flits (one more to contain the physical address) and a response packet (an acknowledgment) of 2 flits.

These characteristics are all shown in the timing diagrams in Figures 5.6 and 5.7, that summarize what happens when an instruction running on the processor $\langle 5, 3 \rangle$ produces a cache miss. Figure 5.6 shows the flow of the request, while Figure 5.7 shows the return of the response to the originating tile. The timing diagrams are obtained by using the trace files of the Tilera clock-accurate simulator.

The model in Equation 5.1 is therefore extended to correctly match the behavior

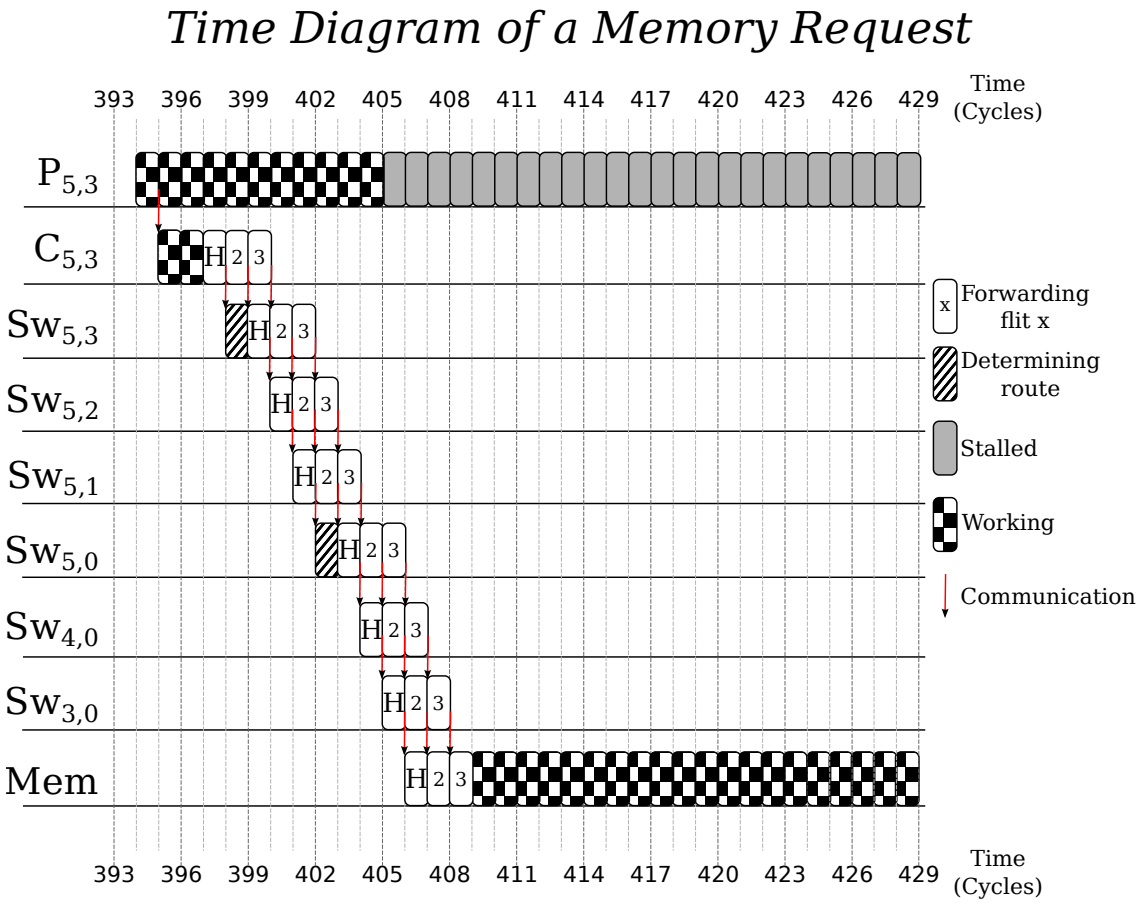


Figure 5.6: Time diagram of the dispatch of a memory read request in the Tile*Pro*64.

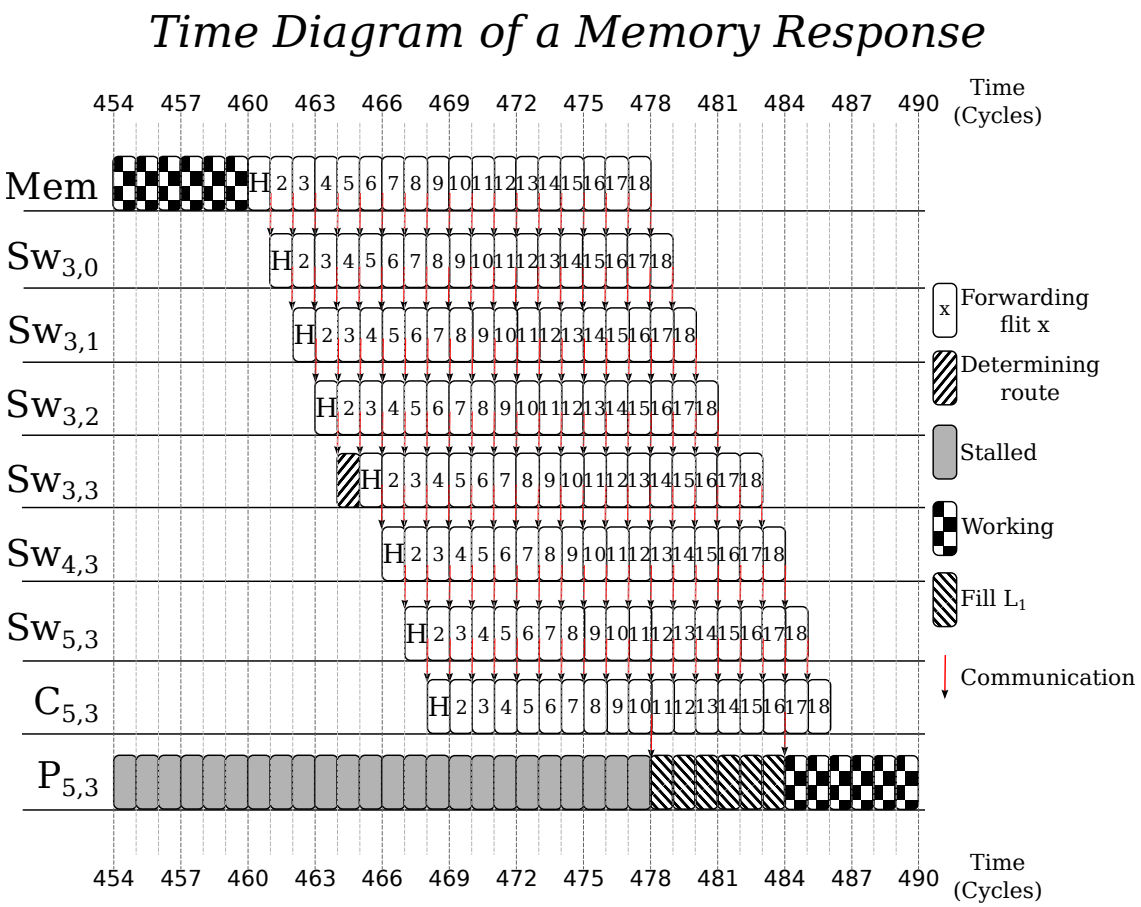


Figure 5.7: Time diagram of the dispatch of a memory read resp. in the Tile*Pro*64.

of the Tile*Pro*64 interconnection. In Equation 5.7, we evaluate the total network latency that a processor pays when a cache line is read or written: starting from the network latency for the request and the response (5.4 and 5.5, respectively), we calculate the total L_{net}. Notice that $m_{req} + m_{resp}$ have been substituted with their value, that is 21 both for memory reads and writes. In addition, while the path for requests is different w.r.t the one for responses, the total number of hops is the same. Nevertheless, the request requires two direction change, so the additive factor is 2, while the response takes only one direction change resulting in the addition of 1. The distance has been divided in d_y and d_x, to easily identify the number of vertical and horizontal hops, respectively; the distances are reported in Table 5.1 for each node. Columns 1 and 2 uses a negative value in d_x because the number of horizontal hops is 0, and the packet do not change direction, so we need to remove the additional routing costs. Given the difference between request and responses, we used the averaged value here.

$$L_{net}^{req} = (d_x + d_y - 1) + (m_{req}) + 2 \tag{5.4}$$

$$L_{net}^{resp} = (d_x + d_y - 1) + (m_{resp}) + 1 \tag{5.5}$$

$$L_{net} = 2(d_x + d_y - 1) + (m_{req} + m_{resp}) + 3 + k \tag{5.6}$$

$$L_{net} = 2(d_x + d_y - 1) + (21) + 3 + k \tag{5.7}$$

Vertical Hops : d_y

Y \ X	0	1	2	3	4	5	6	7
0	1	1	1	1	1	1	1	1
1	2	2	2	2	2	2	2	2
2	3	3	3	3	3	3	3	3
3	4	4	4	4	4	4	4	4
4	5	5	5	5	5	5	5	5
5	6	6	6	6	6	6	6	6
6	7	7	7	7	7	7	7	7
7	8	8	8	8	8	8	8	8

Horizontal Hops : d_x

Y \ X	0	1	2	3	4	5	6	7
0	1	-1.5	-1.5	1	1	2	3	4
1	1	-1.5	-1.5	1	1	2	3	4
2	1	-1.5	-1.5	1	1	2	3	4
3	1	-1.5	-1.5	1	1	2	3	4
4	1	-1.5	-1.5	1	1	2	3	4
5	1	-1.5	-1.5	1	1	2	3	4
6	1	-1.5	-1.5	1	1	2	3	4
7	1	-1.5	-1.5	1	1	2	3	4

Table 5.1: Distance table (d_y and d_x) used to evaluate L_{net}.

Finally, a further factor k is added to Equation 5.7. This will be used to "adjust" L_{net} when comparing it with the values obtained by using the cycle-accurate simulator. The reason is that the L_{net} obtained by the simulator is calculated *at the processor level*, evaluating the *stall time*. The timing diagrams shows that the stall time is slightly different from the real network latency, because the processor enter the stall *some cycles after* the requests (it stall only when the data is required), and exit the stall *some cycles before* the complete reception of the response (it start as soon as the required word is available).

Equation 5.7 has been verified by mean of a hand-made benchmark run on the cycle-accurate simulator. The benchmark consist of a single thread, placed on the selected core, that issue load requests to the first memory controller. The program produces a high number of cache miss by linearly reading an array. After each load, a proper code is executed, using the loaded value, to ensure the processor stall. Notice that in this benchmark the time spent by the processor between two cache miss is irrelevant, as the memory will always serve a message at a time. Cache coherency has been disabled in this test, so that a cache miss immediately produce a read request, without contacting the Home node (that would have produced a different message pattern across the network). The benchmark results are reported in Table 5.2, along with the estimated L_{net} by using $k = -4$. Having an error always below 1 clock cycle, we can safely confirm that Equation 5.7 correctly model the network latency.

Table 5.2: Simulated and Estimated L_{net}. Times are in clock cycles.

Node		L_{net}		Error	Node		L_{net}		Error
X	Y	Simulated	Estimated		X	Y	Simulated	Estimated	
0	0	21.25	22	0.75	4	0	21.36	22	0.64
0	1	24.12	24	0.12	4	1	24.43	24	0.43
0	2	26.24	26	0.24	4	2	26.27	26	0.27
0	3	28.16	28	0.16	4	3	28.34	28	0.34
0	4	30.23	30	0.23	4	4	30.31	30	0.31
0	5	32.33	32	0.33	4	5	32.30	32	0.30
0	6	34.18	34	0.18	4	6	34.07	34	0.07
0	7	36.08	36	0.08	4	7	36.33	36	0.33
1	0	16.24	17	0.76	5	0	23.40	24	0.60
1	1	18.19	19	0.81	5	1	26.48	26	0.48
1	2	20.32	21	0.68	5	2	28.31	28	0.31
1	3	22.23	23	0.77	5	3	30.44	30	0.44
1	4	24.22	25	0.78	5	4	32.37	32	0.37
1	5	26.37	27	0.63	5	5	34.39	34	0.39
1	6	28.09	29	0.91	5	6	36.18	36	0.18
1	7	30.04	31	0.96	5	7	37.34	38	0.66
2	0	16.28	17	0.72	6	0	25.49	26	0.51
2	1	18.25	19	0.75	6	1	28.35	28	0.35
2	2	20.17	21	0.83	6	2	30.34	30	0.34
2	3	22.26	23	0.74	6	3	32.41	32	0.41
2	4	24.19	25	0.81	6	4	34.33	34	0.33
2	5	26.20	27	0.80	6	5	36.33	36	0.33
2	6	28.09	29	0.91	6	6	38.17	38	0.17
2	7	30.11	31	0.89	6	7	40.65	40	0.65
3	0	21.24	22	0.76	7	0	27.34	28	0.66
3	1	24.16	24	0.16	7	1	30.31	30	0.31
3	2	26.29	26	0.29	7	2	32.35	32	0.35
3	3	28.36	28	0.36	7	3	34.31	34	0.31
3	4	30.32	30	0.32	7	4	36.40	36	0.40
3	5	32.31	32	0.31	7	5	38.31	38	0.31
3	6	34.30	34	0.30	7	6	40.09	40	0.09
3	7	39.04	36	3.04	7	7	40.72	42	1.28

Node	L_{net}		Error
	Simulated	Estimated	
Average	29.19	29.25	0.06

5.5.1 Under Load Latency

Now that we know L_{net}, we need to find a way to model the mesh, and in particular the possible switch contention, inside the Queueing Network model. Starting from the results already presented in the previous chapter, we have three different choices:

a) Model the network as a network of queues, one for each switch, and a proper class-based routing to mimic the low-level behavior of the mesh; this approach tends to be excessively complex, as we would have a queue to model each switch, a large number of task classes and class-based routing on each queue.

b) Approximate the behavior of the mesh using the (multi)bus model, with a single, possibly parallel, queue and thus manually limiting the number of requests that the network support concurrently; the queue can also be used to model the network delay; nevertheless we saw that each node have a different L_{net}, so we need to apply an average value in this case.

c) Approximate the behavior of the mesh using the crossbar model: all the processors can issue requests concurrently; in this case we model the delay of the network for each request by coupling each processor with a queue that will represent its connection to the memory, so that we are able to model per-node L_{net}, at the cost of an increased number of queues that, however, are fairly simple, with no class-based routing.

Given the complexity of point a, we are more interested in the two approximations and, in particular, in studying *how much* these approximations differ from the real behavior: if we are able to verify that one offers accurate enough results, we could use a much simpler model.

The idea of using a bus (point b) is quite interesting because of its simplicity and also because of the availability of several previous studies on this kind of models. A single bus, with no concurrency at all is indeed a very rough approximation that will probably affect the result negatively. A multi-bus, on the other hand, should be able to capture the amount of concurrent requests of a mesh; however we have the very hard task of estimating the number of servers for the bus. We should also consider that, w.r.t buses, in a wormhole-routed mesh conflicts do not happen at the level of packet (messages) but between flits; this is a further difference in the behavior that affects the amount and type of conflicts and, therefore, the response time we model.

On the other hand we have another approximation, in which conflicts are not modelled at all. Yet we have the freedom of using a different L_{net} for each node, that better fit the distance between the processor and the memory in this architecture.

At the end, however, the real question to be answered is not really *how many conflicts happen on the mesh* but, instead, *how much they affect the under-load system response time*. While these two are strictly related (i.e. each conflict usually increment the average response time of the system), in a complex, closed-loop system

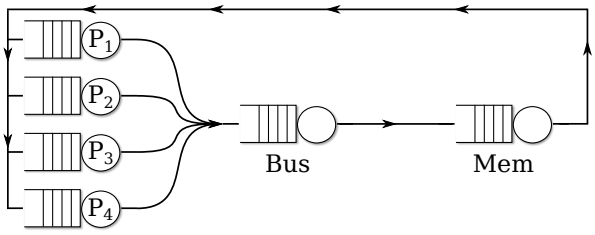


Figure 5.8: Example architecture model.

like our, the two aspects may not be as much related as one could think. In fact, it may happen that a conflict on the interconnection network is completely hidden by the queueing delay of the memory.

Consider the example model in Fig. 5.8, where we have four processors connected to a memory by means of a single bus. Consider constant latencies of $L_{net} = 99$ and $L_{mem} = 100$. The processor queues should have a very low service time, so that the memory subsystem becomes the bottleneck; we selected a $T_p = 1$ with an exponential distribution. Let us say tasks 1 and 2 are generated, respectively, at times $T_1^{P_1} = 1$ and $T_2^{P_2} = 10$ by their respective processors. T_1 is the first to enter the bus and is immediately served: exit the bus at $T_1^{Bus} = 100$, subsequently enter the memory queue and exit at $T_1^{Mem} = 200$. On the other hand, 2 is enqueued in the network; the Bus start handling it at time 100, when task 1 has been completed, and exit at $T_2^{Bus} = 199$. Now, when 2 reaches the memory, it is enqueued because 1 has not yet finished. Its processing by the memory will start at time 200 and will complete at $T_1^{Mem} = 300$. Now, in this example, the queueing time at the bus has been completely masked, as task 2 may enter the memory *any* time between $[101, 200]$ and the final result will always be $T_2^{Mem} = 300$, meaning that we did not pay the conflicts on the bus at all.

A similar behavior also happen at steady state: an analysis of the network[6] shows that, on average, the Bus have an average population of ~ 0.99, and the Memory of ~ 3. This means that we have one task being computed at the Bus queue, two tasks waiting and one computing at the Memory. In practice when a task reaches the Bus, it will find the queue empty and will begin its processing: no conflict happens in the Bus at steady state because the network is self-regulating based on the response of the memory. The response time for a request becomes of $R_q = 400$, for any processor, as one may expect. One can easily verify that by using the crossbar modeling, the steady-state R_q is the same.

This basic example allow us to understand that, if the memory is the component that is slowing down the system, it may not really matter which way the interconnection network is modeled. We already saw that the Memory-Mesh interface has been designed such that the overall bandwidth between the Mesh and the Memory is capable of sustaining the maximum theoretical bandwidth of the Memory. This

[6]Obtained by simulating the network using the JMT[29] simulation tools

alone gives a first insight that the mesh should not be the bottleneck of the system.

From these first considerations, we believe that, for the Tile*Pro*64, a simple yet accurate enough approximation of a mesh may be a crossbar, thus completely neglecting the possible conflicts of the network. We could use the Tile*Pro*64 simulator with specific benchmarks, and compare them w.r.t. queueing network models of Bus and Crossbars. However, at this time we also do not know how to model the Memory and the Processor, so that we would not be able to identify the source of approximation when comparing the results. We needed a way to "put" the Tile*Pro*64 mesh inside a system where the Processor and the Memory behave exactly like their corresponding queues.

We were able to obtain this result by using our EQNSim framework, creating three models where the Memory and the Processors are always modeled in the same way, while the network is defined as follows:

- **A Bus model**, in which the interconnection network is modeled with a single queue with an averaged L_{net} (see Table 5.2).
- **A CrossBar model**, in which each processor is coupled with a queue to model its exclusive connection to the memory; in this case each queue has a different L_{net}, depending on the position of the node in the Mesh.
- **A Mesh model**, in which we interface the Memory and Processor queues to a Network-On-Chip simulator that simulates a Mesh.

The implementation of the Mesh model was possible by using several features of the OMNeT++ framework on which EQNSim is based. We used an already available open-source NoC simulator (HNoCS[27]) that runs within OMNeT++, and interfaced it with our queue modules. The use of an already existing simulator was preferred because it avoided us the excessively long test phase to guarantee that the NoC simulator accurately simulate a Mesh. The Mesh was parameterized to be as much as similar to the target architecture: wormhole, same bandwidth, flit propagation delay and routing algorithm. The only characteristic that we were not able to simulate was the additional 1-clock delay on changing routing direction. We believe, however, that while this slightly affect the L_{net} value, its impact on the conflicts is negligible. To keep the three models comparable, we evaluated a per-node L_{net} on the simulated Mesh with a not loaded system (i.e. a single node working), and used this result to instantiate the L_{net} of the interconnection queue(s) on the other two models. For the Memory queue, we used a constant distribution with L_{mem} = 8clocks. This represents a lower limit on the possible response time of the memory, evaluated by using the maximum theoretical bandwidth of the memory: $52Gbps$, at the frequency of the Tile*Pro*64, means 7.5 bytes per clock cycle. Considering the cache line size of $64Bytes$, the memory takes, at best, 8.5 clock cycles to produce a response to a memory request. The service time of the processor

Model	Processors	L_{net}(clocks)	L_{mem}(clocks)	T_p(clocks)
Mesh	64	22-44	8	50-4000
Bus	64	32	8	50-4000
Crossbar	64	22-44	8	50-4000

Table 5.3: Parameters of the three models compared.

is modelled by an exponential distribution. We simulate different load conditions by changing its average value, within a range of 50 and 4000, where 50 clocks can be considered a realistic lower bound for programs running on that architecture. These parameters are summarized in Table 5.3, while the models are depicted in Figures 5.9, 5.10 and 5.11 using the graphical representation of EQNSim. Fig. 5.9 shows the complexity of the Mesh-based model: each node is composed of a router, and its corresponding core. We use a 10x10 Mesh to easily implement the memory interfaces on the top and bottom rows with the HNoCS routers; however, messages only flows among the inner 8x8 Mesh, mimicking the iMesh behavior. The Bus model (Fig. 5.10) is indeed quite simple, with a square of 8x8 clients connected to the Network Queue (up left). This is connected to the Memory queue (up right), that send results directly to the clients. Finally, Fig. 5.11 shows the Crossbar Model, in which each processor is coupled with its Network queue that is, in turn, connected to the Memory queue.

The simulation results are summarized in Table 5.4 and Figure 5.12. First of all we notice that the Memory is indeed the bottleneck of the architecture. Consider

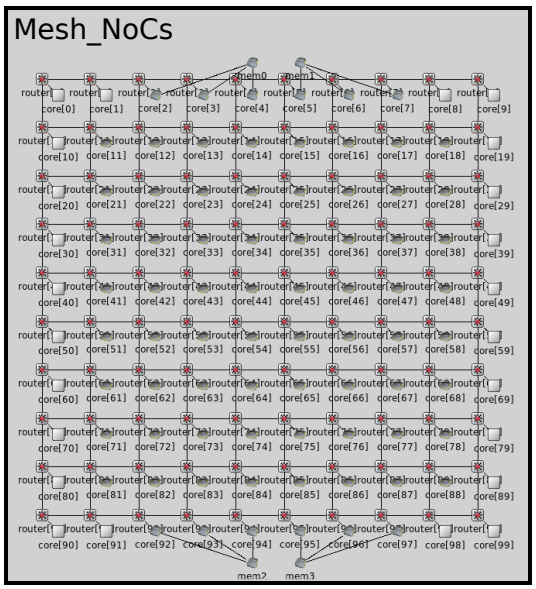


Figure 5.9: Graphical representation of the Mesh model on EQNSim.

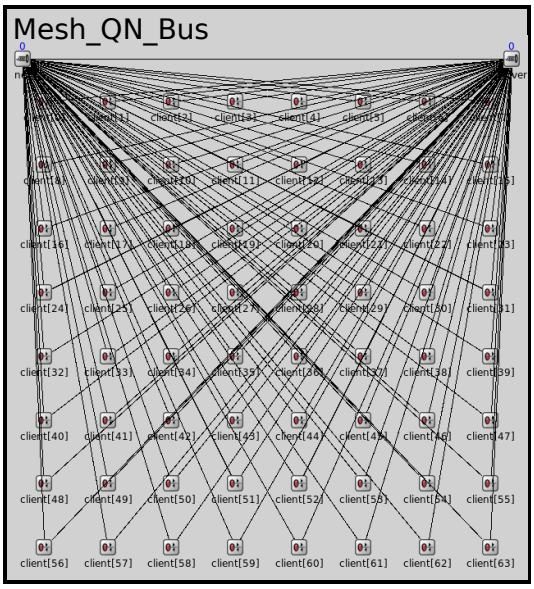


Figure 5.10: EQNSim Bus model.

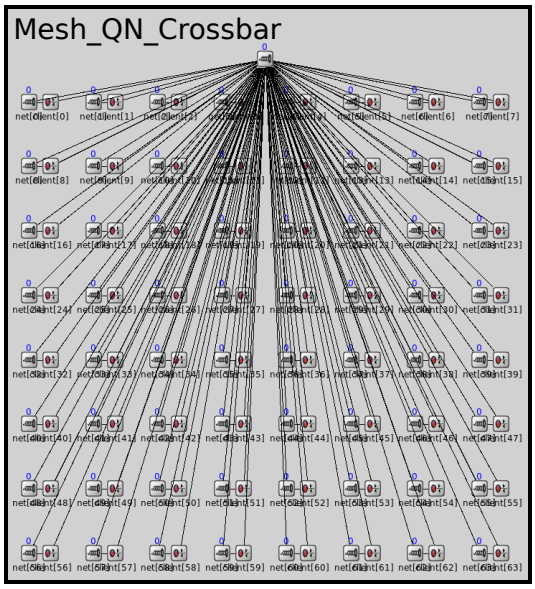


Figure 5.11: EQNSim Crossbar model.

that a not-loaded system would have an $R_q = L_{net} + L_{mem} \simeq 40 clocks$. With $T_p = 50$ the average R_q goes up to $\sim$ 450, that represent more than a ten-fold increase over the original response time. As the T_p increase, the frequency of requests is decreased and the R_q returns to acceptable values: with a $T_p \geq 1000$ we practically obtain the same values of not-loaded systems. From the approximation point of view, as we already expected, the bus introduce an excessive bottleneck, so that we can safely ignore this model for our Queueing Network. On the other hand, the crossbar delivers excellent results, especially with extremely low or high T_p: the $< 3\%$ approximation in the $[50 - 300]$ range is really good for our purpose, while in the $[1000 - 4000]$ we have a slightly higher approximation, within 10%, corresponding to an absolute error of 2-3 clock cycles that is, again, acceptable for our purpose. We are a bit worried about the $[300 - 1000]$ interval, where the error reaches the 27%. It seems that, in this interval, the mesh-induced conflicts affect the resulting response time. In general, a 27% approximation is not acceptable for our model. We should note, however, that these values were obtained with a value of L_{mem} that should be considered a *lower bound*, and not the real average latency of the memory, as it was obtained by considering the maximum theoretical bandwidth of the memory. During the execution of a parallel program this bandwidth will be, for many reasons addressed later in the chapter, only a fraction of the theoretical limit. The graph in Figure 5.13 shows the behavior of the percent error with higher L_{mem}: with 10clocks we already lower the error to a maximum of $\sim$ 13%, and with higher values we can easily bound this error to less than 5%.

Given these considerations, we deem the Crossbar model an accurate enough approximation of the iMesh interconnection network.

T_p	Average R_q			Error		Percent Error	
	Mesh	Crossbar	Bus	Crossbar	Bus	Crossbar	Bus
50	467.7336	461.2997	1989.996	6.433868	1522.263	1.375541	325.4551
100	418.6502	411.7429	1940.942	6.907363	1522.292	1.649913	363.619
150	368.4983	362.2851	1890.861	6.213117	1522.363	1.686064	413.1262
200	320.0667	312.1018	1840.171	7.964893	1520.105	2.488511	474.9338
250	269.2603	263.0031	1791.825	6.257229	1522.564	2.323858	565.4618
300	220.0449	213.2575	1738.633	6.787326	1518.588	3.084519	690.1268
350	172.979	163.0053	1693.006	9.973715	1520.027	5.765853	878.7351
400	135.0604	115.2616	1643.309	19.79882	1508.249	14.65924	1116.722
450	106.8769	80.54745	1590.937	26.32945	1484.06	24.6353	1388.57
500	87.25117	63.46506	1541.733	23.78611	1454.481	27.26165	1667.005
550	74.37144	54.92414	1496.977	19.4473	1422.606	26.14888	1912.839
600	66.04933	50.45271	1448.025	15.59662	1381.976	23.61359	2092.339
650	60.91022	47.61659	1397.757	13.29363	1336.847	21.82496	2194.782
700	57.08377	45.87084	1348.963	11.21293	1291.879	19.64294	2263.128
750	54.36241	44.6865	1299.159	9.675915	1244.796	17.79891	2289.81
800	52.29279	43.66689	1246.425	8.625899	1194.132	16.49539	2283.55
850	50.93849	42.91707	1197.703	8.021418	1146.765	15.74726	2251.274
900	49.51538	42.52503	1143.915	6.99035	1094.4	14.11753	2210.221
950	48.46683	41.99369	1106.365	6.473146	1057.898	13.35583	2182.726
1000	47.65006	41.69449	1044.466	5.955567	996.816	12.49855	2091.951
1500	43.66096	39.80172	552.7655	3.859236	509.1045	8.839101	1166.041
2000	42.28185	39.05176	177.1227	3.230088	134.8408	7.639421	318.9096
2500	41.7978	38.86385	89.96244	2.933954	48.16464	7.019398	115.2325
3000	41.35286	38.62825	67.71589	2.724604	26.36303	6.588672	63.75142
3500	40.98316	38.552	58.61975	2.431159	17.6366	5.932093	43.03377
4000	41.13613	38.4112	53.2243	2.724929	12.08817	6.624175	29.38576

Table 5.4: Simulation Results with $L_{mem} = 8$ clocks.

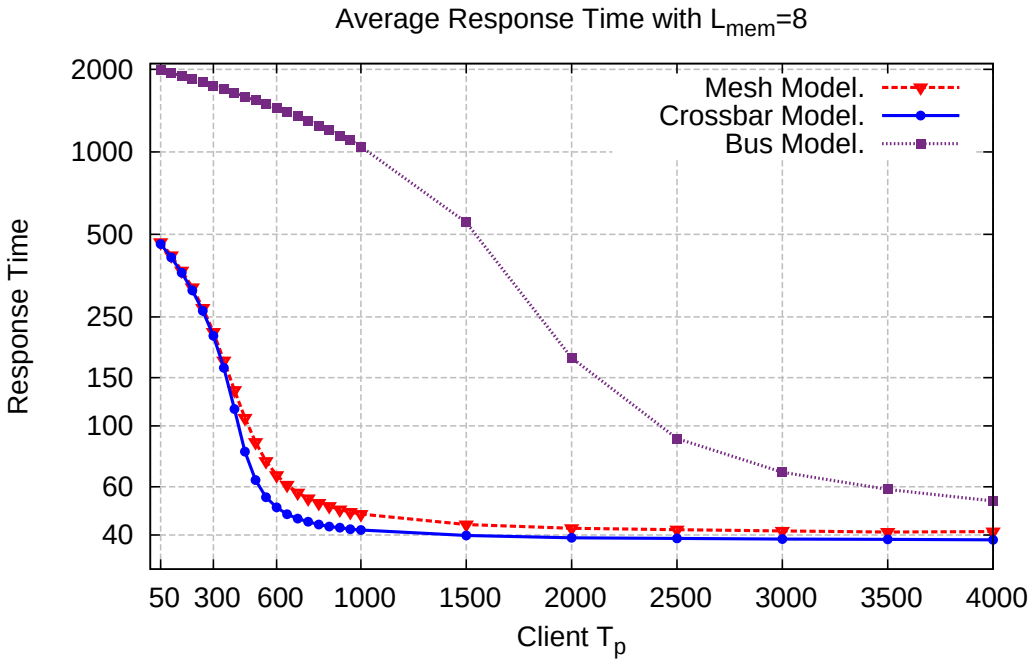


Figure 5.12: Graphical comparison of the average R_q for the selected models.

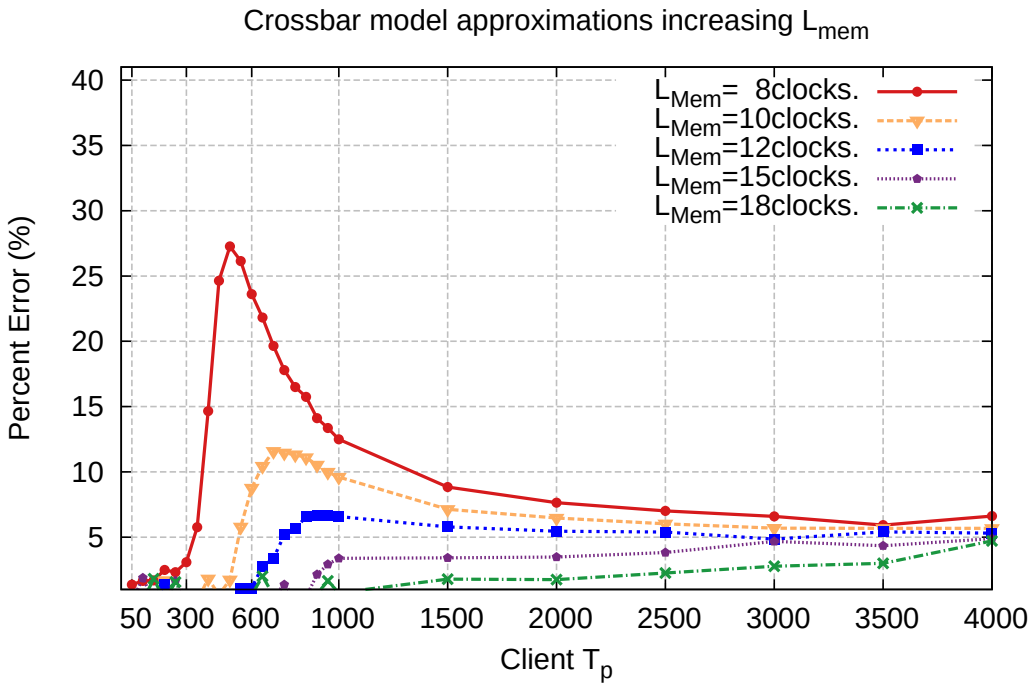


Figure 5.13: Graphical comparison of the percent error of the Crossbar model w.r.t the Mesh increasing L_{mem}.

5.6 Memory Subsystem

The Tile*Pro*64 chip is connected to four independent memory controllers. The presence of multiple memory interfaces is not really a problem from the modeling point of view, as even the original model by Bhandarkar[36] considered multiple memories by using a queue for each memory, and a proper probability distribution of tasks among the different memories. We will call these probabilities p_{mc1}, p_{mc2}, p_{mc3}, and p_{mc4}. What we need to model is the response time of a single memory, so that we are able to parameterize the memory queues. Bhandarkar's model considered a simple sequential memory, in which all memory operations took the same amount of time, regardless of the type (i.e. load or store) and of the system load (i.e. amount of enqueued requests).

Unfortunately, memory technology saw an important increase in complexity to address the requirements driven by processor evolution. As a result, current memory modules may exhibit very different access times depending on the sequence and timing of issued requests. Their behavior is so complex that cycle-accurate simulators *of the memory subsystem* have been developed in the last years[146], to study in detail the behavior of a memory under various loads. The interested reader can refer to Wang's PhD thesis[178] for a detailed description of current ram technologies, their access protocol and the implementation details of a memory controller. We will just highlight the most important points required for our modeling purpose.

Working with cache lines

Memories works on single addresses, composed of a limited number of bits (64bits for the DDR2 memories of the Tile*Pro*64), certainly smaller than the size of a cache line. Therefore, at least from a theoretical point of view, the reading or writing of a cache line should require the issuing of several memory operations.

In practice, however, given the fact that since several years all the processors uses caches, and thus require memory operations on entire cache lines, memory commands have been optimized to issue single request working on entire cache lines. This is possible by exploiting some form of parallelism and buffering inside the memory (such that the whole cache block is read at once, and then buffered to be sent throughout the 64-bit interface using multiple clock cycles). This, together with pipelined requests (i.e. while the memory is producing the sequence of 64-bit words, another address selection operation can be issued), allow us to safely model the memory operation time as the selection of the first address, T_{mem}, plus a fixed latency T_{mc} required by the memory controller to receive the other words from the memories and prepare the response packet to be sent: we have that $L_{mem} = T_{mem} + T_{mc}$.

It is important to notice that, because of pipelining, it is also safe to assume that the memory queue has a service time equal to T_{mem}, as it can start processing the next cache block without waiting the production of all the data. The Tile*Pro*64 simulator traces shows a constant $T_{mc} = 41$ clocks.

5.6.1 Memory Read Service Time

We start with the analysis of T_{mem} in the case of memory read operations. Then, we will generalize the study in case of mixed read and write operations. At this point of the model, we know that the network interconnection can be efficiently modeled as a crossbar. Using **EQNSim** we had two ways:

- Study the memory system alone, as with done with the network interconnection: take the original Bhandarkar's model and substitute the memory queue with a cycle-accurate memory simulator such as DRAMSim2.
- Study the memory system *and* the interconnection network, using for example HNoCS for the interconnection network and DRAMSim2 for the memory, and compare this simulation with an extended model, in which the interconnection network is modeled using the results of the previous section.

Given the previous results, and towards the definition of a complete model, the two approaches are equally good. However, both of them are technically difficult to realize because the memory simulator requires a trace of memory addresses to behave correctly. While this is possible because the extensibility of the simulation environment, it requires the production of these trace files, starting from real (or at least realistic) programs. The final resulting model would still need to be parameterized using the cycle-accurate Tile*Pro*64 simulator to model the exact timings of the architecture. We therefore considered the direct use of the the cycle-accurate Tile*Pro*64 simulator. This was made possible because we are able to write specific programs with a well-defined constant T_p. In this scenario we know how to model (in the QN world) two of the three major entities of the model: *processor* and *network*, so that the modeling and verifying effort is just confined to the single unknown component. All the tests presented in this section will therefore compare the Tile*Pro*64 simulator, where we explicitly disabled the **automatic cache coherence** so that memory requests are directly sent to the memory controller, w.r.t the QN-based model executed on EQNSim.

Open Page Policy

For both historical and performance reasons, memory chips divide the physical address in several parts: **row**, **column** and **bank** (at least). Different "banks" correspond to independent memory arrays. *Rows* and *Columns* are more interesting, because they affects the *response time* of the memory: for technical reasons, row and column selections cannot be executed concurrently, so that we pay a time of $T_{mem} = t_{RCD}$[7] $+ T_{CL}$[8] to select a specific word from the memory. To reduce the memory latency, memory controllers exploit the so-called *open-page policy*: once a

[7]RAS to CAS delay, indicate the time to select a row
[8]CAS latency, indicate the time to select a column once the row has been selected

row is selected, multiple data belonging to the same row can be gathered by just changing the column. If the controller keep a row open, subsequent requests result in $T_{mem} = T_{CL}$. Of course this is possible only if the addresses belong to the same row (called *page hit*). A common way of exploiting locality is to map the lower part of the physical address to the column, so that consecutive physical addresses belong to the same row. This allow to exploit the open-page policy in basically all the algorithms that generate streams of consecutive addresses.

The Tile*Pro*64, as most of the current architectures, implement the open-page policy, so we created a small number of benchmarks to test its behavior. In our tests the cycle-accurate simulator of the architecture was of great importance, as it allowed us to measure the memory service time T_{mem} for each individual request. We created two different sequential benchmarks, written in assembler code, to test the behavior of the memory subsystem:

1. **load_linear**, reported in Listing 5.1, in which the code perform multiple loads from a contiguous memory area by linearly scanning an array.

2. **load_sparse**, reported in Listing 5.2, in which the code perform multiple loads scattered among a memory area by reading data from a linked list carefully allocated such that each element belong to a different, random location of a large memory area.

```
addi r5, %0, 0     #  N
addi r4, zero, 0   #  i
addi r10, %1,  0   #  b
.START_bench:
  slte r6, r5, r4 # N<=i
  bnz r6, .END
  addi r7, zero, 0   #  j
  addi r11, zero, N_NOPS
  STARTJ:
    slte r12, r11, r7
    bnz r12, .ENDJ

    REPEAT_10( addi r9, r9, 1)
    REPEAT_10( addi r9, r9, 1)
    addi r7, r7, 1 # j++
    j .STARTJ
  .ENDJ
  addi r8, r4, 0   # i
  s2a r8, r8, r10 # b+4i
  lw r9, r8        # load
  addi r4, r4, 16 # i+=16
  j .START_bench
```

Listing 5.1: load_linear Benchmark.

```
addi r4, %0, 0   # *list


.START_bench:
  lw r5, r4 # curr->next

  addi r7, zero, 0   #  j
  addi r11, zero, N_NOPS
  .STARTJ:
    slte r12, r11, r7
    bnz r12, .ENDJ
    addi r9, r5, 0
    REPEAT_10( addi r9, r9, 1 )
    REPEAT_10( addi r9, r9, 1 )
    addi r7, r7, 1 # j++
    j .STARTJ
  .ENDJ:

  # curr=curr->next
  addi r4, r5, 0
  # check for curr=NULL
  bnz r4, .START_bench
```

Listing 5.2: load_sparse Benchmark.

If we run the two benchmarks on the simulator, we obtain two very different average memory service time:

$$\begin{aligned} T_{mem}^{linear} &= 9.9313\tau \\ T_{mem}^{sparse} &= 33.18011\tau \end{aligned} \tag{5.8}$$

A detailed inspection shows exactly what we expected: with the **load_linear** benchmark, most memory reads take 8 clocks, that is exactly T_{CL}, as the requests find the row already open (*page hit*); conversely, with **load_sparse** most operations takes 34 clocks, corresponding to $t_{RCD} + T_{CL}$ because the requests require a change of the open row (*page miss*). More precisely, in Figure 5.14 we show the various response times and the corresponding percent of requests. The result shows that in the linear case $\sim 91\%$ of the requests find page already open, while in the sparse case $\sim 93\%$ of them produce a page miss, requiring to change the row.

This effect could become a big problem in our queueing network, as the difference is so high that we would need to model the effect of different memory access patterns. Fortunately (for our purpose), the *open page policy* does not work very well on a parallel program, where each core generates its own stream of memory requests: in this case, even if each stream would exploit the open page policy, when the requests are mixed together, the memory controller is not able to take advantage of the open pages. To test this condition we created a semi-parallel version of the previous code, in which multiple independent threads running the same benchmark are created and allocated on different cores. The resulting memory service times breakdown using 64 cores is reported in Figure 5.15: we can easily notice that the effects of the *open page policy* completely vanish, resulting in most requests taking exactly 34 cycles in both cases. To understand when this effect is triggered on the **load_linear** benchmark we report the average T_{mem} when varying the number of cores used in Figure 5.16: we can observe that with 2 cores the time is already increased and with only 4 cores the two benchmarks exhibit the same average T_{mem}.

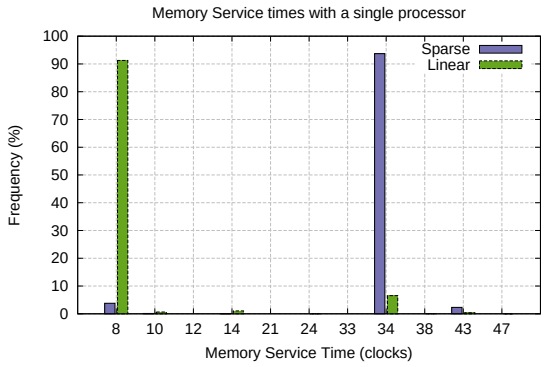


Figure 5.14: Detailed breakdown of T_{mem} using a single benchmark core.

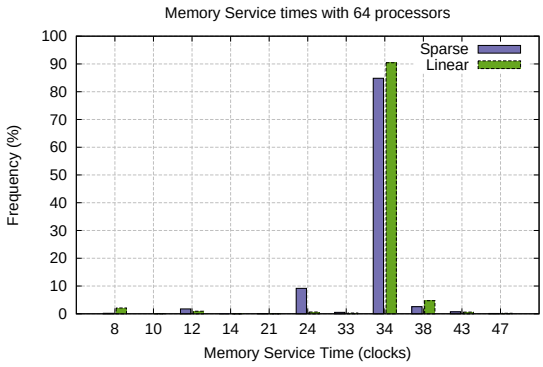


Figure 5.15: Detailed breakdown of T_{mem} using 64 benchmark cores.

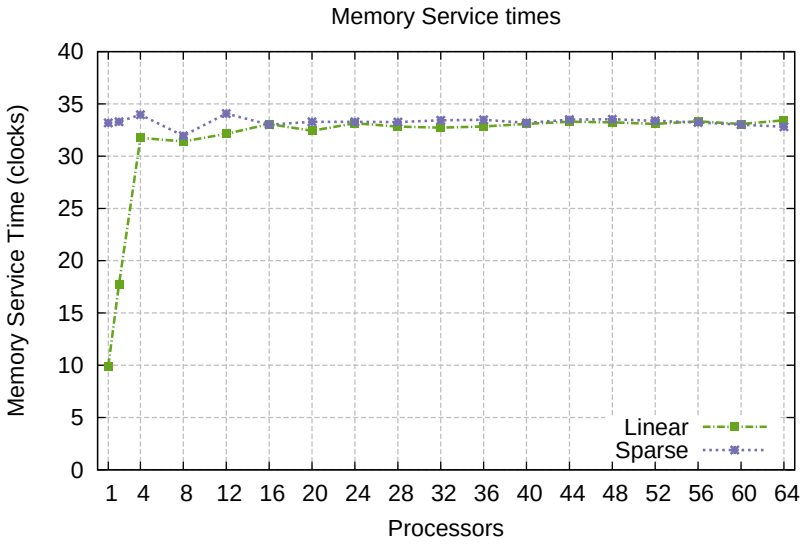


Figure 5.16: Average T_{mem} varying the number of cores generating memory requests.

It should be noted that for this benchmark we selected a number `N_NOPS` particularly high (resulting in $T_p = 3015$ cycles) so that the memory controller receives a single request at a time. We verified this by checking the number of enqueued requests at the memory controller, that is always zero, even with 64 cores.

For this reason we believe that neglecting the possible performance improvement of the open page policy, thus using $T_{mem} = t_{RCD} + T_{CL}$ consist in a good approximation for our model.

Request reordering and parallelism inside the controller

In the previous benchmarks we deliberately selected an `N_NOPS` so that the memory was never a bottleneck. This is not a realistic case for our model, as our objective is indeed evaluating the response time *when the memory is a bottleneck*. So we decreased the waiting time between two requests to test the memory under load. For the sake of conciseness, we show only the results for **load_sparse**. We selected three different decreasing values of T_p, to understand the behavior of T_{mem} under load. The results reported in Figure 5.17 and Table 5.5 are pretty clear: as the frequency of requests increase, the response time of the memory T_{mem} tend to decrease.

A breakdown for the requests in the 64 cores experiment (Figure 5.18) show an intriguing result: the number of plain page miss is decreased, in favor of service times of 24 and 12 clocks that, however, are still marked by the simulator as "page miss". We are still changing the row but we are able to do it faster, probably because of some pipeline effects between the selection of an address and the reply from the memories.

	$T_p = 40$		$T_p = 1015$		$T_p = 3015$	
	L_q	T_{mem}	L_q	T_{mem}	L_q	T_{mem}
1	0	33.91437	0	33.87897	0.000552	33.18011
2	0.004517	33.93857	0.036969	27.42596	0.001427	33.29444
4	0.038971	33.75643	0.028406	30.40966	0.001195	33.95565
8	0.849501	27.69459	0.01323	34.02548	0.015322	31.94242
12	2.195512	21.00406	0.024923	33.94668	0.004226	34.07601
16	4.007253	17.66907	0.045807	33.79771	0.023232	33.00857
20	7.070376	15.85322	0.072103	33.62934	0.021096	33.28639
24	9.680906	15.18303	0.113479	33.31475	0.030062	33.28132
28	13.02022	14.49752	0.239494	32.24164	0.042859	33.25115
32	16.30971	14.10587	0.438312	30.73988	0.052501	33.42837
36	20.07768	13.97966	0.588089	29.90958	0.053674	33.4772
40	24.17332	13.78302	0.810481	28.40889	0.084061	33.18345
44	27.3199	13.67006	1.236569	25.84654	0.081238	33.49642
48	30.91879	13.63092	1.465879	24.53571	0.082645	33.53439
52	34.73994	13.51555	1.830985	22.8638	0.108236	33.38213
56	38.89917	13.51747	2.155002	21.47939	0.133312	33.21996
60	42.92958	13.49963	2.533383	20.18774	0.177057	33.01201
64	46.56375	13.48873	2.918729	19.20154	0.211984	32.80809

Table 5.5: Average T_{mem} and memory queue length varying T_p and the number of cores. Times in cycles.

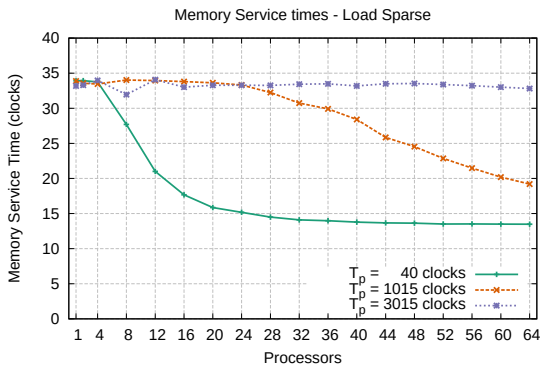


Figure 5.17: Average T_{mem} varying the number of cores and T_p.

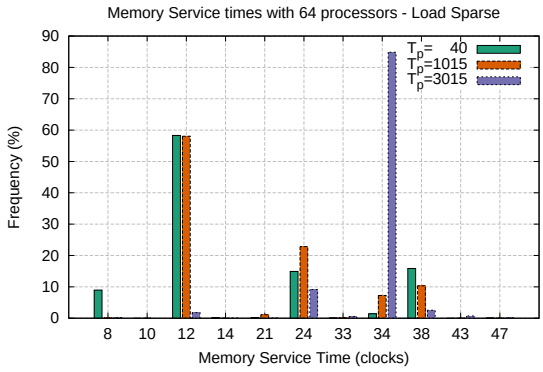


Figure 5.18: Detailed breakdown of T_{mem} using 64 cores with $T_p = 40$clocks.

Furthermore, with $T_p = 40$ clocks, we find some requests that actually generate a *page hit*, obtaining a $T_{mem} = 8$ clocks. This is probably due to the *reordering engine* of the memory controller, that, in presence of several requests, try to order them as much as possible to guarantee page hits.

The final result is that the memory is faster if many requests are enqueued. In particular, measuring the request queue of the memory controller (that keeps all the requests that still need to be processed) seems very important to determine T_{mem}: in Table 5.5 we can see that a value of $L_q \simeq 0.8$ result in a $T_{mem} \simeq 27 - 28$ cycles, for both $T_p = 40$ and $T_p = 1015$. The same apply for $L_q \simeq 2.1$, that correspond to a $T_{mem} \simeq 21$ cycles. We confirmed the effect by further investigating this correlation: Table 5.6 report a broader range of L_q values, with the corresponding T_{mem} and core configurations for the two faster frequencies ($T_p = 3015$ is irrelevant as the queue is always empty).

L_q	$T_p = 40$			$T_p = 1015$		
	L_q	T_{mem}	Cores	L_q	T_{mem}	Cores
~ 0.5	0.527595	30.2587	7	0.588089	29.90958	36
~ 0.8	0.849501	27.69459	8	0.810481	28.40889	40
~ 1.2	1.182808	25.73326	9	1.236569	25.84654	44
~ 1.8	1.873849	22.37465	11	1.830985	22.8638	52
~ 2.1	2.195512	21.00406	12	2.155002	21.47939	56

Table 5.6: Correlation between L_q and T_{mem}.

A natural way of modeling this is by using a **load dependent queue** [112] for the memory, that allow us to select different values of T_{mem} depending on the actual load of the queue. Of course, the idea seems pretty good, but we need to find the correct values of T_{mem} for each possible queue size.

By parsing the trace files of the Tile*Pro*64 simulator we are able to identify, for each memory request, its T_{mem} and the number of enqueued requests. We therefore gathered these statistics by using several runs, with different T_p and number of cores, of the **load_sparse** benchmark, and obtained the average T_{mem} for each possible queue size. The results are reported in Table 5.7 and Figure 5.19. We used these values for our load-dependent memory queue model. The values were first smoothed (as reported in the same Table), mainly for two reasons:

- When designing **EQNSim**, we expected that, modeling a discrete system driven by clock cycles, it would have been helpful to define queueing network elements that worked with the same time unit. We therefore defined a "clock cycle", and each event time of the simulation is rounded to the next clock cycle. While this seemed a good idea in the beginning, now that we work with averaged values, we have to use round values for our load-dependent queue.

L_q	% of Samples	T_{mem} Meas.	T_{mem} Smooth	L_q	% of Samples	T_{mem} Meas.	T_{mem} Smooth
0	26.735	33.16098	33	34	1.218	13.43913	13
1	6.156	26.65501	27	35	1.246	13.39959	13
2	4.358	21.1373	21	36	1.266	13.43915	13
3	3.254	18.05551	18	37	1.287	13.42106	13
4	2.295	16.27781	16	38	1.288	13.45796	13
5	1.631	15.33176	15	39	1.301	13.43534	13
6	1.282	14.80604	15	40	1.326	13.44543	13
7	1.131	14.54571	15	41	1.329	13.42193	13
8	1.089	14.26347	14	42	1.33	13.41958	13
9	1.077	14.03642	14	43	1.333	13.48715	13
10	1.085	13.90968	14	44	1.353	13.48852	13
11	1.06	13.76392	14	45	1.368	13.43211	13
12	1.035	13.65461	14	46	1.392	13.43136	13
13	1.037	13.51876	14	47	1.412	13.47264	13
14	1.073	13.32687	13	48	1.422	13.40865	13
15	1.091	13.12141	13	49	1.274	13.53869	13
16	1.084	13.2854	13	50	1.068	13.66938	13
17	1.089	13.31197	13	51	0.824	13.80125	13
18	1.114	13.35151	13	52	0.588	13.93621	13
19	1.123	13.38625	13	53	0.377	14.06716	13
20	1.091	13.42446	13	54	0.22	14.09796	13
21	1.032	13.49279	13	55	0.116	14.29843	13
22	1.034	13.47754	13	56	0.056	14.90873	13
23	1.054	13.43892	13	57	0.023	14.60017	13
24	1.111	13.41633	13	58	0.009	14.71524	13
25	1.138	13.38652	13	59	0.002	15.58015	13
26	1.152	13.42235	13	60	0.001	16.77586	13
27	1.156	13.42171	13	61	0	15.21052	13
28	1.159	13.40428	13	62	0	12	13
29	1.135	13.47351	13	63	0.147	31.16372	13
30	1.146	13.44923	13	64	0.124	31.73716	13
31	1.164	13.42909	13	65	0.088	31.07267	13
32	1.183	13.47575	13	66	0.049	27.65798	13
33	1.185	13.40266	13	67	0.033	24.70453	13

Table 5.7: Measured values of T_{mem} and the smoothed value used for the load-dependent queue, for different queue sizes.

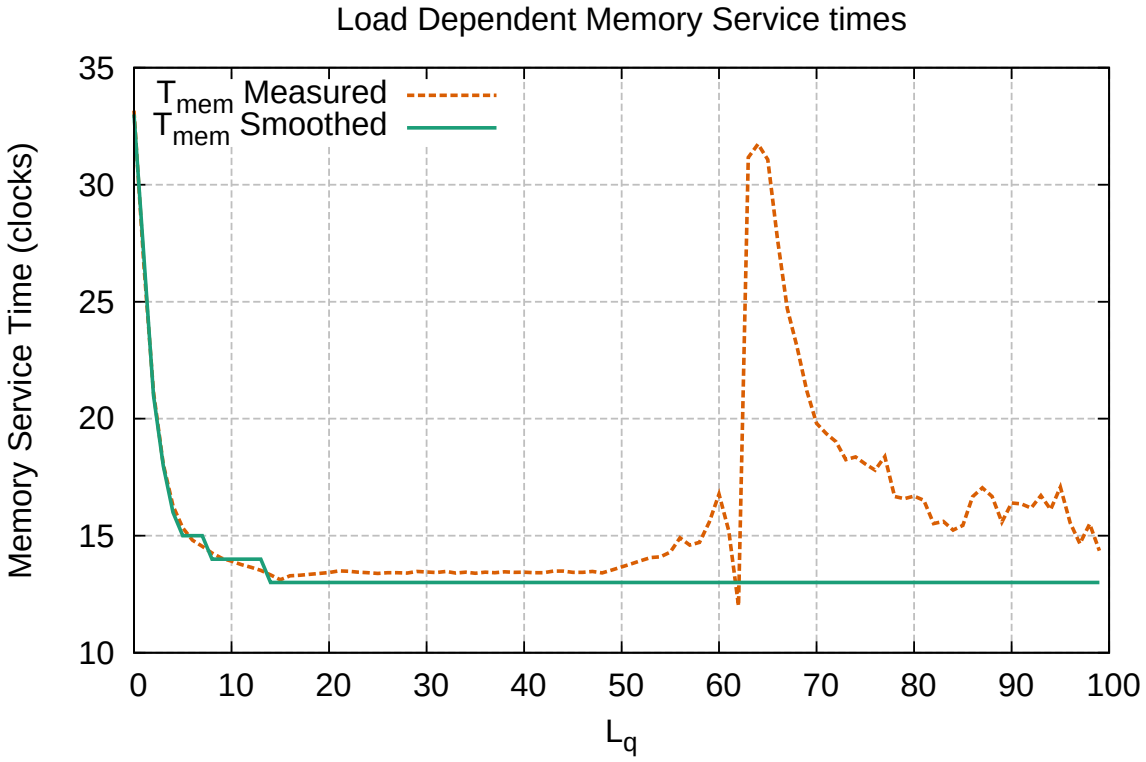


Figure 5.19: Measured T_{mem} varying L_q and the smoothed value for the model.

- With L_q higher than 50, the number of samples used for computing the average is minimal, greatly below 1%, meaning that we have a small number of events. The amount of data collected is not sufficient to consider these numbers a correct average of the behavior, and would give us a wrong approximation of T_{mem}; in fact, of the spike that happen with $L_q = 63$ is obviously misleading, as it is very unlikely that it represent the real behavior. We decided to cut-off the value of T_{mem} to the previous value of 13, that produces a much more reasonable behavior.

Now that we have the service time parameters for the load-dependent memory queue, we are able to build a queueing network that should model the entire Processor-Memory subsystem and evaluate the steady state values w.r.t the data obtained by the Tile*Pro*64 simulator. This will allow us to confirm if the idea of using a load-dependent queue is a feasible way of modeling the particular behavior of the Tile*Pro*64 memories. The problem of finding values for T_{mem} that holds for a generic program will be addressed later.

In Figure 5.20 we show the resulting queueing network, with a queue per processor to model the interconnection network delay, a load-dependent queue for modeling T_{mem}, plus a second queue to model T_{mc}.

In Table 5.8, Table 5.9 and Table 5.10 we summarize the results for T_{mem} and L_q

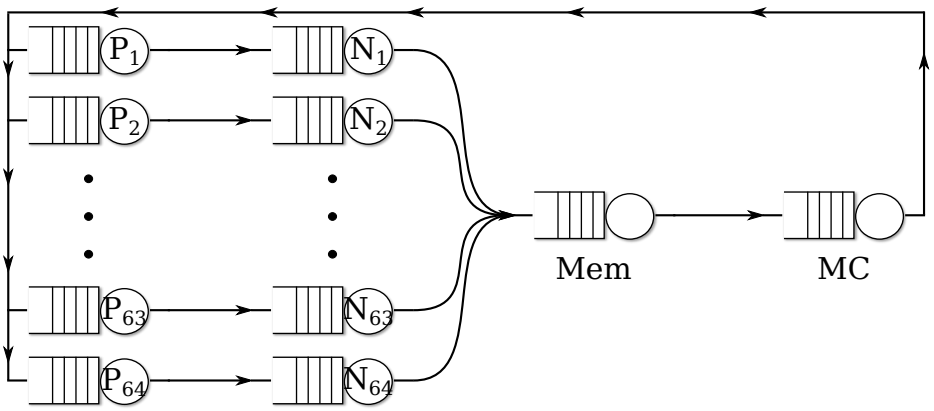


Figure 5.20: Queueing network model with a load dependent memory queue and independent network queues per core.

of the Tile*Pro*64 simulator and our model, using the previously studied T_p values. With the very high T_p of ~ 3000 cycles (Table 5.8), the model works very well, giving values very similar to the ones of the Tile*Pro*64. Still, we see that the average L_q of the model is a bit higher than the one of the memory controller queue, resulting in small variations of T_{mem}.

Decreasing T_p to ~ 1000 cycles (Table 5.9), the model is still working very well, especially on T_{mem}, that is always within 3 cycles of difference w.r.t the architectural value. We notice, again, the problem on L_q that this time is more evident (at the end of the table we have a difference of 2 on the average queue length), but this does not affect so much T_{mem}.

Finally, the most interesting results are with $T_p = 40$ cycles, when the memory is a real bottleneck. Here we start paying too much for the difference between the two L_q: with 64 cores the difference is of 8. This problem is highlighted mostly in the middle of the table, when the values of L_q are between $[5, 20]$ and the difference in T_{mem} that can reach 4 cycles, corresponding on an error of almost 20% percent. This is a quite large difference, that we expect will affect the precision of the predicted R_q. Fortunately, with very high L_q values T_{mem} on the Tile*Pro*64 memory stabilize at 13 clocks, so the difference in L_q does not affect T_{mem}.

The difference on the estimated and the measured L_q is probably due to the approximations on the network interconnection: the rationale behind using a cross-bar was to avoid a complex network because conflicts can just be "postponed" on the memory interface; however, with a load-dependent queue, we recognize that this *may* affect the precision of the model because the queueing network produces higher values of L_q (as all the conflicts are handled at the memory point); in the cases studied the approximation seems good enough; however if more accuracy is need, this surely represent a point in which the QN model can be further extended.

Cores	TilePro64		Model		Error		
	L_q	T_{mem}	L_q	T_{mem}	L_q	T_{mem} Abs	T_{mem} Percent
1	0.00	33.18	0.01	33.00	0.01	0.18	0.54
2	0.00	33.29	0.02	33.00	0.02	0.29	0.88
4	0.00	33.96	0.04	33.00	0.04	0.96	2.81
8	0.02	31.94	0.08	33.00	0.07	1.06	3.31
12	0.00	34.08	0.13	33.00	0.12	1.08	3.16
16	0.02	33.01	0.18	32.92	0.16	0.09	0.26
20	0.02	33.29	0.23	32.86	0.21	0.43	1.28
24	0.03	33.28	0.28	32.81	0.25	0.47	1.43
28	0.04	33.25	0.33	32.74	0.29	0.51	1.54
32	0.05	33.43	0.39	32.63	0.34	0.80	2.40
36	0.05	33.48	0.45	32.51	0.39	0.96	2.87
40	0.08	33.18	0.51	32.38	0.42	0.81	2.43
44	0.08	33.50	0.58	32.21	0.49	1.28	3.83
48	0.08	33.53	0.64	32.02	0.56	1.51	4.51
52	0.11	33.38	0.72	31.82	0.61	1.57	4.69
56	0.13	33.22	0.79	31.59	0.66	1.63	4.91
60	0.18	33.01	0.87	31.34	0.69	1.67	5.06
64	0.21	32.81	0.95	31.06	0.74	1.74	5.31

Table 5.8: Comparison between architecture and model values of L_q and T_{mem} with $T_p = 3015$ clocks. Times in cycles.

Cores	TilePro64		Model		Error		
	L_q	T_{mem}	L_q	T_{mem}	L_q	T_{mem} Abs	T_{mem} Percent
1	0.00	33.88	0.03	33.00	0.03	0.88	2.59
2	0.04	27.43	0.06	33.00	0.02	5.57	20.32
4	0.03	30.41	0.10	33.00	0.07	2.59	8.52
8	0.01	34.03	0.24	32.85	0.23	1.18	3.46
12	0.02	33.95	0.41	32.68	0.38	1.27	3.75
16	0.05	33.80	0.58	32.30	0.53	1.50	4.44
20	0.07	33.63	0.77	31.78	0.70	1.85	5.50
24	0.11	33.31	0.99	31.09	0.88	2.22	6.67
28	0.24	32.24	1.24	30.19	1.00	2.05	6.36
32	0.44	30.74	1.50	29.15	1.06	1.59	5.16
36	0.59	29.91	1.80	27.94	1.21	1.97	6.59
40	0.81	28.41	2.11	26.64	1.30	1.77	6.23
44	1.24	25.85	2.43	25.31	1.20	0.54	2.09
48	1.47	24.54	2.78	23.96	1.31	0.57	2.33
52	1.83	22.86	3.14	22.64	1.31	0.22	0.96
56	2.16	21.48	3.52	21.42	1.36	0.06	0.28
60	2.53	20.19	3.93	20.26	1.40	0.07	0.35
64	2.92	19.20	4.37	19.23	1.46	0.02	0.12

Table 5.9: Comparison between architecture and model values of L_q and T_{mem} with $T_p = 1015$ clocks. Times in cycles.

Cores	Tile*Pro*64		Model		Error		
	L_q	T_{mem}	L_q	T_{mem}	L_q	T_{mem} Abs	T_{mem} Percent
1	0.00	33.91	0.23	33.00	0.23	0.91	2.70
2	0.00	33.94	0.48	33.00	0.47	0.94	2.77
3	0.02	32.05	0.82	32.73	0.80	0.68	2.13
4	0.04	33.76	1.18	32.26	1.14	1.50	4.44
8	0.85	27.69	3.08	23.28	2.23	4.42	15.96
12	2.20	21.00	5.13	16.92	2.93	4.09	19.45
16	4.01	17.67	7.99	14.85	3.98	2.82	15.96
20	7.07	15.85	11.51	14.02	4.44	1.83	11.56
24	9.68	15.18	15.03	13.45	5.35	1.73	11.40
28	13.02	14.50	18.67	13.00	5.65	1.49	10.30
32	16.31	14.11	22.59	13.00	6.28	1.11	7.84
36	20.08	13.98	26.57	13.00	6.50	0.98	7.01
40	24.17	13.78	30.50	13.00	6.32	0.78	5.68
44	27.32	13.67	34.51	13.00	7.19	0.67	4.90
48	30.92	13.63	38.44	13.00	7.52	0.63	4.63
52	34.74	13.52	42.43	13.00	7.69	0.52	3.81
56	38.90	13.52	46.33	13.00	7.43	0.52	3.83
60	42.93	13.50	50.32	13.00	7.39	0.50	3.70
64	46.56	13.49	54.27	13.00	7.71	0.49	3.62

Table 5.10: Comparison between architecture and model values of L_q and T_{mem} with $T_p = 40$ clocks. Times in clock cycles.

Response time prediction accuracy

In the previous section we analyzed how to parameterize T_{mem} and T_{mc} for our model; now, it is finally time to test the entire model, in order to check its level of accuracy in predicting R_q. We report the numerical values with various T_p and cores on Table 5.11. We are able to model the most difficult behavior (with $T_p = 40$) with an average error of $\sim 5.7\%$ and *at most* of $\sim 12.5\%$. For the sake of completeness we also report a graphical comparison of the three values of T_p in Figure 5.21, where we can appreciate the accuracy of the modeled R_q function. We are very happy of the model of Figure 5.20, that is able to estimate the R_q of our parallel application with a sufficient level of accuracy.

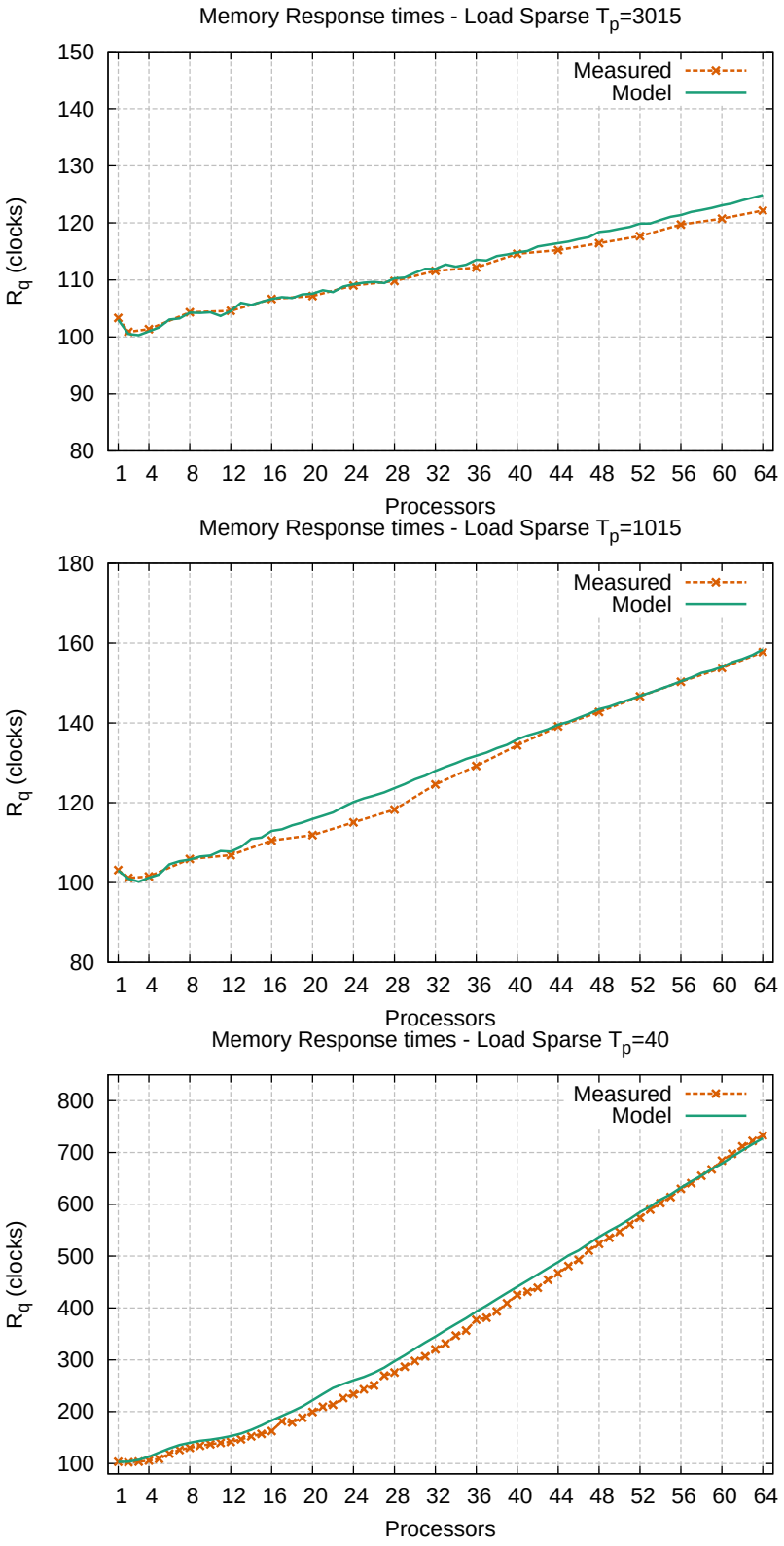


Figure 5.21: Comparison between architecture and model values of R_q. Times in clock cycles.

	3015				1015				40			
	R_q		Error		R_q		Error		R_q		Error	
	Real	Model	Abs	%	Real	Model	Abs	%	Real	Model	Abs	%
1	103.31	103.00	0.31	0.30	103.11	103.00	0.11	0.11	103.25	103.00	0.25	0.24
2	100.88	100.50	0.38	0.38	101.14	100.96	0.18	0.18	102.43	103.44	1.01	0.99
4	101.36	100.97	0.38	0.38	101.52	101.22	0.29	0.29	105.62	112.97	7.35	6.96
8	104.31	104.25	0.05	0.05	105.91	105.75	0.16	0.15	130.11	139.96	9.84	7.56
12	104.57	104.55	0.02	0.02	106.87	107.76	0.90	0.84	141.59	152.77	11.17	7.89
16	106.64	106.65	0.01	0.01	108.18	112.93	4.75	4.39	162.55	182.82	20.27	12.47
20	107.16	107.58	0.42	0.39	111.88	115.94	4.06	3.63	199.41	222.03	22.62	11.34
24	109.02	109.24	0.22	0.20	115.08	120.17	5.09	4.43	233.87	260.37	26.50	11.33
28	109.80	110.27	0.47	0.43	118.27	123.65	5.38	4.55	275.30	297.20	21.89	7.95
32	111.57	111.93	0.36	0.32	124.58	127.97	3.40	2.73	320.06	344.83	24.77	7.74
36	112.15	113.51	1.36	1.21	129.21	131.77	2.56	1.98	377.08	393.07	15.99	4.24
40	114.58	114.82	0.25	0.21	134.37	135.84	1.47	1.09	425.10	440.92	15.82	3.72
44	115.22	116.43	1.21	1.05	139.13	139.54	0.41	0.30	466.98	488.49	21.51	4.61
48	116.43	118.41	1.98	1.70	142.74	143.45	0.71	0.50	523.69	537.23	13.54	2.58
52	117.67	119.88	2.21	1.88	146.65	146.77	0.12	0.08	574.32	584.90	10.58	1.84
56	119.69	121.37	1.68	1.41	150.30	150.42	0.13	0.08	629.98	632.16	2.18	0.35
60	120.74	123.07	2.33	1.93	153.77	154.01	0.23	0.15	684.28	678.68	5.60	0.82
64	122.16	124.87	2.71	2.22	157.73	158.41	0.67	0.43	732.87	727.31	5.56	0.76

Table 5.11: Comparison between architecture and model values of R_q. Times in clock cycles.

5.6.2 Memory Write Service Time

Now that we are able to model memory read operations and their service time, it is time to analyze the behavior of the memory w.r.t write operations. In basically all the previously studied models, there is rarely a distinction in the type of operations for the memory queue, but we still need to check if this approximation is sufficient for our architectural model.

We took one of the previous tests, **load_linear**, and modified the code to issue a store after each load. In this more realistic benchmark, that we called **store_linear**, the memory receives both read and write operations. The assembler code is basically untouched, as reported in Listing 5.3, except for the store that write the register value back in the memory.

As in the previous study, we start with an analysis of the service time when the memory is not a bottleneck, and we compare it with the service time for the loads. For the sake of clarity, we now denote with T^R_{mem} and T^W_{mem} the memory service times for read and write operations, respectively. We take the old value of $T_p \simeq 3000$, run the benchmark and calculate, starting from the simulation traces, the average service time of *only* memory writes. We graphically show the result in Figure 5.22, where we can notice two important things:

1. With the $T_p \simeq 3000$, the **store_linear** benchmark produce a greater pressure on the memory w.r.t **load_linear**, that translates in a decrease of the average

```
addi r5, %0, 0     #  N
addi r4, zero, 0   #  i
addi r10, %1,  0   #  b
.START_bench:
  slte r6, r5, r4 # N<=i
  bnz r6, .END
  addi r7, zero, 0   #  j
  addi r11, zero, N_NOPS
  STARTJ:
    slte r12, r11, r7
    bnz r12, .ENDJ
    REPEAT_10( addi r9, r9, 1)
    REPEAT_10( addi r9, r9, 1)
    addi r7, r7, 1 # j++
    j .STARTJ
  .ENDJ
  addi r8, r4, 0   # i
  s2a r8, r8, r10 # b+4i
  lw r9, r8        # load
  sw r8, r9        # store
  addi r4, r4, 16 # i+=16
  j .START_bench
```

Listing 5.3: store_linear Benchmark.

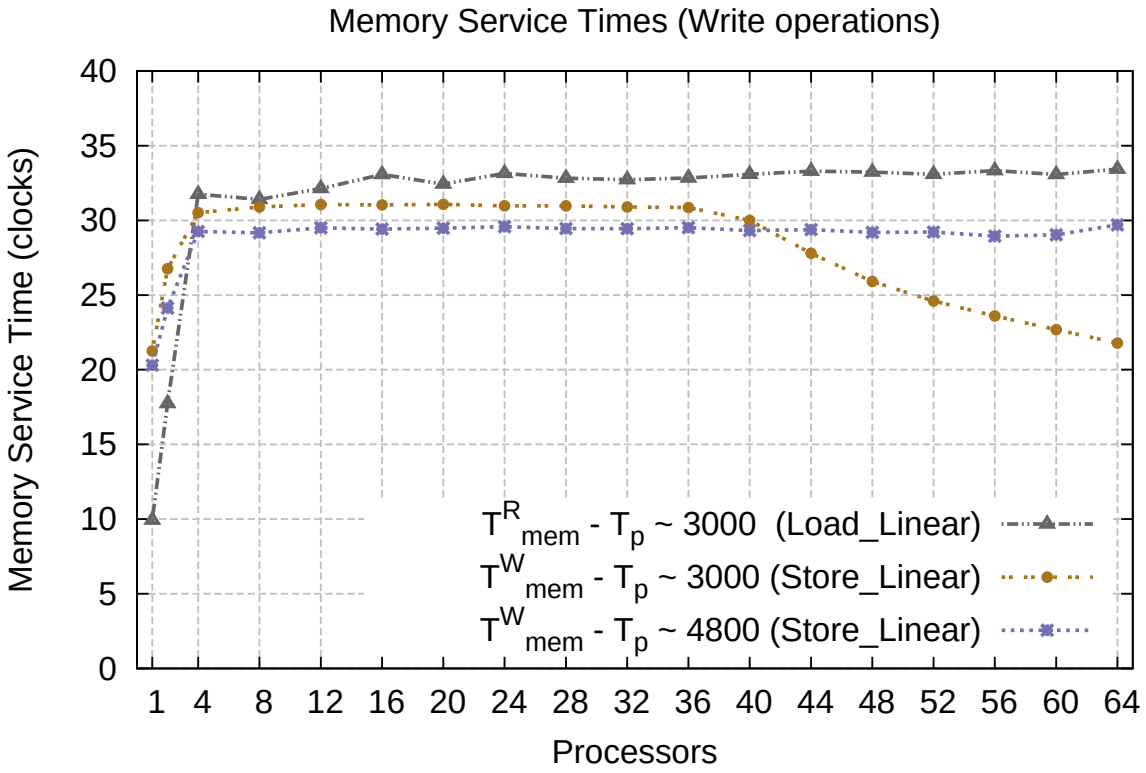


Figure 5.22: Average T^W_{mem} varying the number of cores generating memory requests.

service time with a high number of cores. This is indeed to be expected, as we keep the same T_p but double the amount of requests per time (we now have one read and one write request every T_p), so we just need to further increase the T_p to obtain the baseline; a value of $T_p \simeq 4800$ clocks seems to be sufficient.

2. Memory service times for the two kind of requests seems slightly different: at the beginning, when the open page policy should ensure page hits, the write operation seems slower; then, from 4 cores on, when we basically have only page misses, the write operation seems slightly faster than the read one.

The second aspect surely deserve a further investigation, so we compared the service time per request of **load_linear** and **store_linear** for the 1 and 4 cores benchmarks. The two breakdowns, reported in Figures 5.23 and 5.24 show some important behaviors that explains the results:

- First of all, we notice that for the "common" cases, $T^W_{mem} = T^R_{mem}$: both have a service time of 8 clocks and 34 clocks for page hit and miss, respectively.

- With a single core, however, the store benchmark is not able to fully exploit the open page policy, as only 16% of the requests generate a clean page hit. The reason is that, in **store_linear**, even with a single worker, the memory works

with two different address streams: one for the sequence of load operations, and one for the sequence of store operations. Mixing them together, as in this case, generate an increase in page misses when the two streams works with sufficiently distant accesses. For this reason we have that $\sim 40\%$ of the write requests generate a page miss. This is not really a problem in our model, as we already decided to handle all the memory requests as *page miss*.

- Mixing read and write operations may also increase its service time: in Figure 5.23 we have a lot of requests ($\sim 38\%$) that takes slightly more than the time of a page hit. The Tile*Pro*64 simulator marks them as "hit after read", meaning that, in fact, switching from read to write operations causes a further overhead. This may be a problem, especially if we have the same behavior with reads (i.e. "hit/miss after write"), so we probably need to evaluate again the parameters for T^R_{mem} on an heterogeneous environment with read and write requests.

- With four cores the two breakdowns are more similar: both takes a limited fraction of page hits, while the large part of requests incur in a page miss. However, for the store case, we also have a significant percent of requests that takes 24 or 12 cycles. We already saw this behavior on the load benchmark, when the memory was a bottleneck, and motivated it with the possibility of pipelining requests to the memory. In this case one would think that it is not possible to pipeline requests, as the memory is not a bottleneck, so the probability of having multiple request should be negligible. In fact, however, what usually happen is that each processor send *almost concurrently* a read and a write request, because of the behavior of the L2 cache. With this level of concurrency it is indeed highly probable that the memory controller receive two requests (of different type) and is able to partially overlap the second with the first, explaining the behavior of Figure 5.24. This should not pose any problem, as the behavior should be completely captured by the load-dependent queue.

Under load behavior

Decreasing the value of T_p (Figure 5.25) we obtain results qualitatively similar to the ones previously seen with **load_sparse**: as the amount of enqueued requests at the memory controller increase, the average service time tend to decrease.

However, the obtained values are numerically different. This means that the values of the load-dependent queue modeling the memory should be different in case of write operations. In the next section we will handle the problem of modeling a memory queue that behave differently when executing read and write operations; for the moment we are just interested in gathering the values to parameterize the load-dependent queue as previously done with the read-only operations.

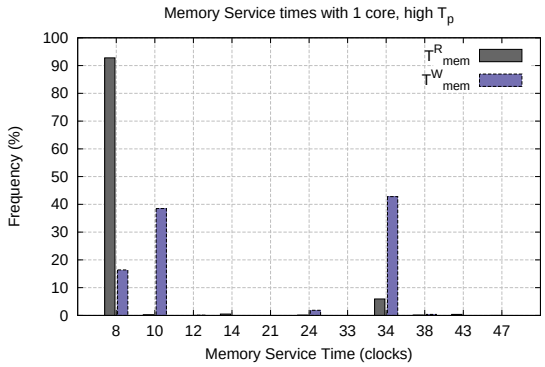


Figure 5.23: Detailed breakdown of T_{mem} using a single benchmark core.

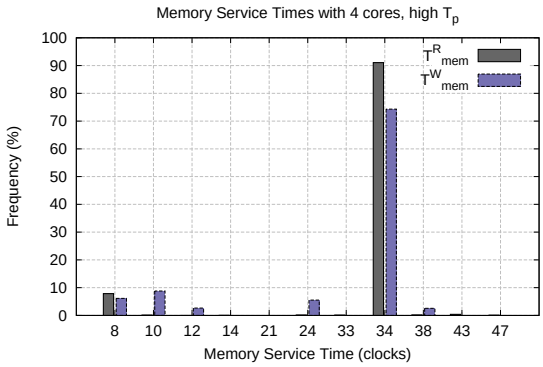


Figure 5.24: Detailed breakdown of T_{mem} using 4 benchmark cores.

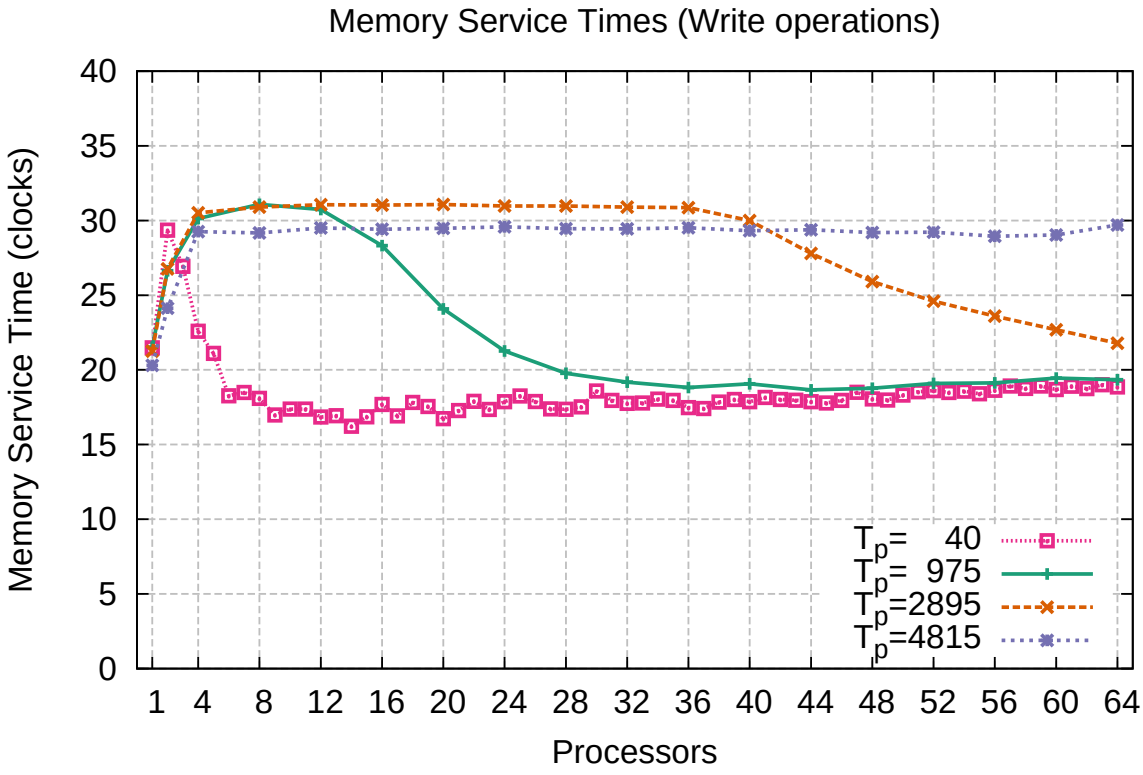


Figure 5.25: Average T^W_{mem} varying the number of cores generating memory requests and the T_p.

Parameterizing the Memory Queue for write requests

We used the same methodology previously applied, analyzing trace files of several runs of the benchmark, to obtain average values for T^R_{mem} and T^W_{mem} depending on the size of the memory controller queue. The results are shown in Table 5.12 for reads and Table 5.13 for writes. The tables clearly show that the behavior is quite different w.r.t the previously obtained, especially for T^R_{mem} that is actually higher than the previous of several clocks. As hypothesized, executing reads after writes have a cost, that is actually quite high and require a new parameterization of the queue. It is important to note that now the maximum queue size is increased, as we produce up to two requests every T_p, ending with a queue size of at most $2*64 = 128$ requests.

We can model the different behavior of read and writes by defining a load-dependent queue with class-based service times, i.e. a queue that have different service times also depending on the class that is being served. This way, we can introduce two different classes for read and write requests, and use the data of Tables 5.12 and 5.13 to define the service times. Otherwise, knowing the amount of read and write requests, we can use a weighted mean of T^R_{mem} and T^W_{mem} for each queue size.

For the sake of conciseness we are not testing, here, the precision of this part of the model. This is mainly because we are not able to measure an R_q on the benchmark, as the write operations are non-blocking and happen in different times w.r.t the related store instructions, because of the caches. We will address this point in the next section, by modeling the behavior of the L2 cache of the tiles and therefore being able to precisely characterize the flow of requests issued by the processor.

Table 5.12: Measured values of T^R_{mem} used for the load-dependent queue, for different queue sizes.

Lq	% of Samples	T^R_{mem}	Lq	% of Samples	T^R_{mem}	Lq	% of Samples	T^R_{mem}	Lq	% of Samples	T^R_{mem}	Lq	% of Samples	T^R_{mem}
0	8.89	39.33	25	0.52	25.51	50	0.77	25.26	75	0.82	24.05	100	0.77	24.38
1	2.55	35.31	26	0.53	25.63	51	0.78	25.33	76	0.82	24.00	101	0.76	24.45
2	1.54	31.42	27	0.61	27.06	52	0.76	24.80	77	0.81	23.95	102	0.76	24.48
3	1.19	30.40	28	0.55	25.62	53	0.82	25.67	78	0.80	23.93	103	0.75	24.56
4	0.87	28.53	29	0.55	25.44	54	0.77	24.64	79	0.80	23.94	104	0.75	24.54
5	0.75	28.52	30	0.57	25.41	55	0.78	24.55	80	0.80	23.91	105	0.75	24.59
6	0.68	28.20	31	0.60	25.68	56	0.78	24.39	81	0.79	23.89	106	0.76	24.59
7	0.73	28.93	32	0.63	25.94	57	0.80	24.65	82	0.78	23.96	107	0.75	24.54
8	0.67	27.84	33	0.64	25.86	58	0.80	24.54	83	0.78	24.07	108	0.75	24.53
9	0.66	27.66	34	0.62	25.11	59	0.81	24.62	84	0.79	24.20	109	0.75	24.53
10	0.63	26.98	35	0.63	25.06	60	0.81	24.53	85	0.79	24.25	110	0.74	24.51
11	0.61	26.92	36	0.65	25.30	61	0.83	24.71	86	0.80	24.23	111	0.75	24.53
12	0.60	26.54	37	0.67	25.42	62	0.84	24.79	87	0.81	24.29	112	0.74	24.56
13	0.60	26.26	38	0.68	25.31	63	0.86	24.93	88	0.80	24.23	113	0.74	24.67
14	0.59	25.73	39	0.67	25.15	64	0.82	24.42	89	0.80	24.21	114	0.73	24.65
15	0.64	26.71	40	0.67	24.99	65	0.80	24.10	90	0.80	24.28	115	0.70	24.75
16	0.56	25.30	41	0.68	24.96	66	0.79	23.97	91	0.80	24.32	116	0.66	24.69
17	0.54	25.13	42	0.69	24.99	67	0.79	23.96	92	0.80	24.41	117	0.61	24.66
18	0.53	25.02	43	0.71	25.00	68	0.80	24.00	93	0.80	24.43	118	0.54	24.55
19	0.59	26.67	44	0.73	24.96	69	0.81	24.06	94	0.79	24.39	119	0.45	24.42
20	0.53	25.63	45	0.74	24.96	70	0.81	24.10	95	0.79	24.42	120	0.37	24.18
21	0.51	25.56	46	0.75	24.97	71	0.81	24.10	96	0.79	24.33	121	0.29	23.95
22	0.51	25.28	47	0.76	25.00	72	0.82	24.12	97	0.78	24.32	122	0.22	23.63
23	0.59	27.00	48	0.76	25.05	73	0.81	24.16	98	0.78	24.35	123	0.17	23.25
24	0.53	25.67	49	0.76	25.22	74	0.82	24.09	99	0.77	24.37	124	0.13	22.90

Table 5.13: Measured values of T^W_{mem} used for the load-dependent queue, for different queue sizes.

Lq	% of Samples	T^R_{mem}	Lq	% of Samples	T^R_{mem}	Lq	% of Samples	T^R_{mem}	Lq	% of Samples	T^R_{mem}	Lq	% of Samples	T^R_{mem}
0	7.65	32.67	25	0.47	18.66	50	0.69	18.15	75	0.87	18.42	100	0.85	18.66
1	2.36	28.47	26	0.47	18.66	51	0.69	18.19	76	0.87	18.46	101	0.84	18.63
2	1.89	24.33	27	0.48	18.57	52	0.71	18.22	77	0.87	18.47	102	0.83	18.70
3	1.16	21.75	28	0.49	18.69	53	0.72	18.27	78	0.86	18.45	103	0.82	18.73
4	0.93	20.70	29	0.50	18.59	54	0.74	18.30	79	0.86	18.52	104	0.83	18.76
5	0.75	20.45	30	0.51	18.71	55	0.75	18.38	80	0.86	18.45	105	0.83	18.70
6	0.69	20.33	31	0.53	18.71	56	0.76	18.22	81	0.85	18.36	106	0.84	18.74
7	0.67	20.68	32	0.54	18.38	57	0.76	18.24	82	0.84	18.40	107	0.84	18.76
8	0.67	20.23	33	0.54	18.25	58	0.76	18.27	83	0.83	18.39	108	0.84	18.71
9	0.66	20.14	34	0.55	18.26	59	0.77	18.24	84	0.84	18.43	109	0.84	18.73
10	0.62	19.50	35	0.56	18.10	60	0.78	18.25	85	0.85	18.46	110	0.83	18.73
11	0.60	19.42	36	0.57	18.20	61	0.80	18.22	86	0.86	18.51	111	0.84	18.73
12	0.59	19.30	37	0.58	18.10	62	0.81	18.22	87	0.87	18.62	112	0.83	18.72
13	0.59	19.24	38	0.59	18.15	63	0.81	18.27	88	0.86	18.40	113	0.83	18.73
14	0.59	18.93	39	0.59	18.14	64	0.81	18.06	89	0.86	18.44	114	0.83	18.83
15	0.59	18.67	40	0.60	18.21	65	0.81	18.10	90	0.86	18.47	115	0.80	18.85
16	0.55	18.80	41	0.60	18.12	66	0.82	18.16	91	0.86	18.48	116	0.76	18.98
17	0.53	18.77	42	0.61	18.16	67	0.82	18.22	92	0.86	18.52	117	0.71	19.06
18	0.50	18.70	43	0.63	18.15	68	0.84	18.33	93	0.86	18.57	118	0.63	19.13
19	0.49	18.52	44	0.65	18.15	69	0.85	18.36	94	0.86	18.59	119	0.54	19.18
20	0.47	18.69	45	0.66	18.16	70	0.86	18.48	95	0.86	18.58	120	0.45	19.32
21	0.47	18.60	46	0.67	18.18	71	0.86	18.43	96	0.86	18.58	121	0.35	19.31
22	0.47	18.83	47	0.67	18.09	72	0.86	18.45	97	0.85	18.58	122	0.27	19.29
23	0.48	18.89	48	0.68	18.12	73	0.87	18.43	98	0.85	18.54	123	0.21	18.97
24	0.48	18.63	49	0.68	18.11	74	0.87	18.45	99	0.85	18.58	124	0.16	18.92

5.6.3 Working with Caches

We complete the memory subsystem analysis by modeling of the behavior of caches, and in particular defining how and when memory requests are generated. In a system with no caches, the behavior is quite simple: when a request is generated by the processor, it is directly sent to the memory. With caches, the behavior is quite more complicate.

Load instructions

Read operations can be modeled quite easily: on a load instruction, the processor will ask the cache for the data and, if the required cache line is not available, a read request is generated. We can easily model the generation of read operations by using the miss probability: a load instruction executed by the processor will generate a read with probability p^l_{miss}. In this case, the processor will stop, waiting for the result from the cache to continue the execution. This is a first approximation, as in reality the processor is free to continue the execution of the following instructions, as long as they do not depend on the load. However, in an in-order processor like the Tile*Pro*64, it is indeed quite usual that a cache miss will stall the processor, as there are no complex mechanisms to advance in the computation.

Store instructions

On the contrary, store operations can be quite difficult to be modeled, for several reasons:

- **Stores are non-blocking**, i.e. the processor can continue the execution without waiting for the completion of the store; this means that, from the processor point of view, it is not much different if the store is handled completely by the cache or forwarded to the memory
- **Store *can* generate cache misses**: if the address to be modified is not already allocated in cache, the cache will first issue a memory read operation to gather the current values from the memory. This is required for correctness reasons, because the cache works only on lines, not single words, and does not know if the program will end modifying all the words of the line.
- Tile*Pro*64 **caches are write-back**, meaning that modified values are not forwarded *immediately* to the memory. In fact, a cache line is written back *only* when selected by the LRU replacement policy to make space for another line.

The first point is just telling us that all the memory operations required to execute a store does directly affect the performance of our program, as the processor is free to continue its execution. Nevertheless, these operations *must* be considered

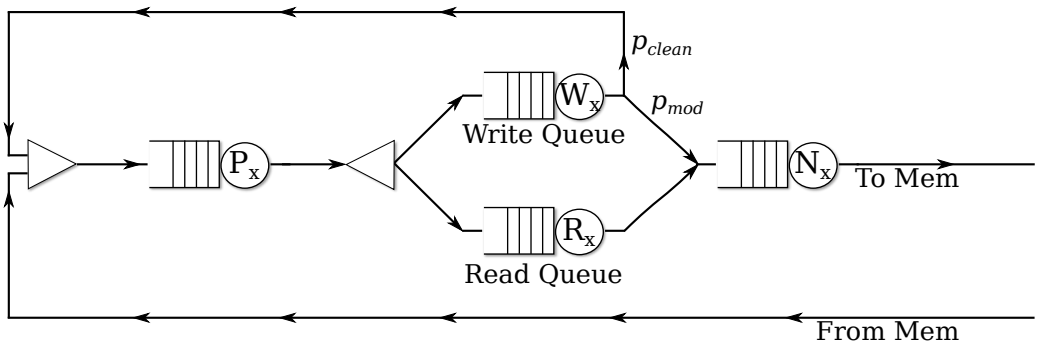


Figure 5.26: Processor-cache subsystem in our queueing network model.

to correctly evaluate the load of the system and, in particular, of the memory queue. For the second point, we just need to define (by using profiled data from a previous execution, for example), the *miss probability* of a store, p^s_{miss}, and generate a *read request* according to that, exactly like the case of load operations.

The third is, indeed, the most difficult characteristic to be modeled. First of all, it tell us that a memory write operation can happen only because of the replacement of a modified cache line. So, to identify the amount of memory write, we should consider the probability of replacing a modified line. Of course, we also need to consider the problem of *when* write operations are issued: one could consider them in T_p, i.e. defining T_p as the time between two memory operations, *regardless* of their type. Yet, this does not correctly model the behavior, as in a real environment a write operation is **always** issued concurrently to a read operation (for the new cache line). If we threat the two requests separately, we would have a lower overall pressure on the memory: for example, we could not have more than 64 requests enqueued (the number of processors), while we already saw by analyzing the Tile*Pro*64 simulator traces that the memory controller queue can easily exceed that value. To issue the two requests concurrently we created the queueing network in Figure 5.26 to model the processor-cache subsystem.

The idea is quite simple: each task generated by the processor queue is split in two different tasks by a fork node: one representing a read, that is always forwarded to the memory, and one representing the "companion" write. Of course, the write is not always generated in the real architecture, so we use a probabilistic routing to:

- send the write task to the memory (p_{mod}): this represent the replacement of a modified cache line and in this way we have two requests that reach the memory;
- send the write task directly to the join node ($p_{clean} = 1 - p_{mod}$): this represent the replacement of a clean cache line, where the write request is not sent to the memory.

In both cases, the semantic of the join node allow the task to be joined and sent back to the processor queue *only* when both of them are back. This way we can easily model the existence of two concurrent requests and, at the same time, keep the model fairly simple. By modifying the routing probability of the write queue, we are able to model different programs, with different amount of write operations.

With this modeling, the processor queue becomes fairly simple to be parameterized, as we just need to find the average time between two memory read requests; these values can be easily gathered by using the processor performance counters we already mentioned in the previous chapter, considering both load and store misses.

5.7 Model Validation

Now that we know how to model every aspect of the memory subsystem we can try to evaluate the accuracy of the queueing network on the previously studied benchmarks. The final model is depicted in Figure 5.27, were we have:

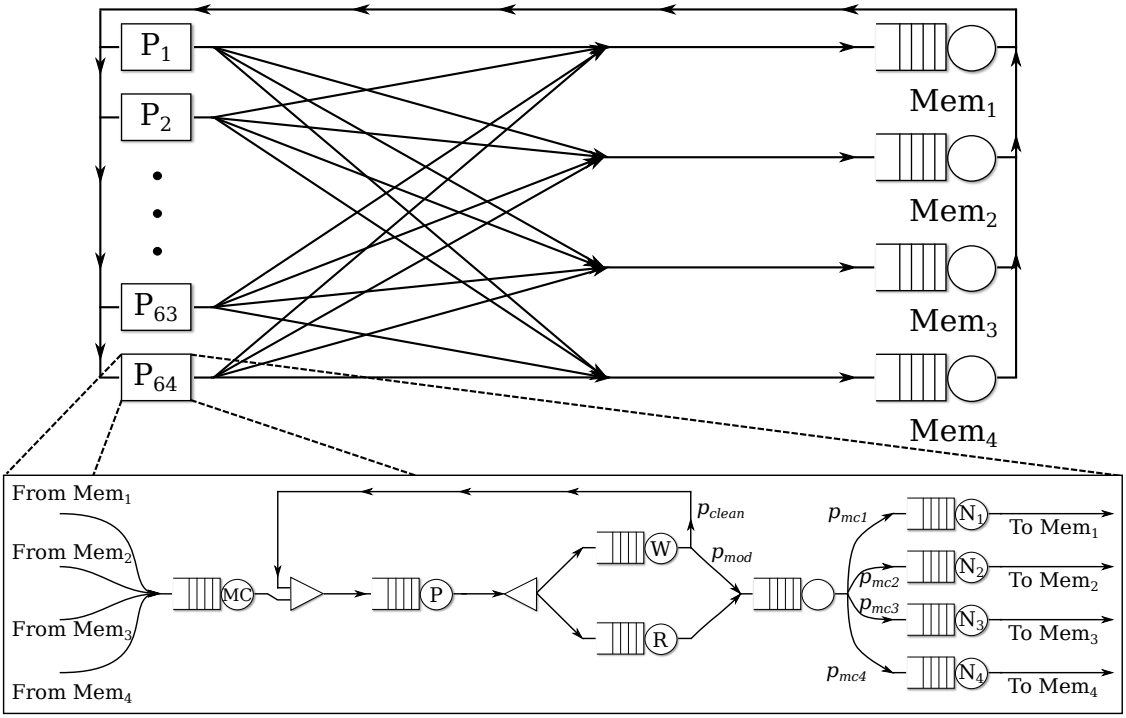
- The **Processor-Cache** subsystem modeled by using several queues and the fork/join nodes, to generate concurrent read and write requests.
- The **Interconnection network** modeled as four crossbars, one per memory interface, in which each processor have a different set of queues to model its specific network latency. The selection of the output queue (i.e. crossbar interface) is done with probabilistic routing.
- The **Memory** modeled as a queue with load-dependent, class-based service times for T_{mem}, followed by a second, simpler queue with constant service times for T_{mc}.

During the experiments with the model, we decided to apply a further approximation: in the current version of **EQNSim** we did not have an implementation for class-based service times; following the rapid prototyping philosophy, we decided to adapt the current load-dependent queue to behave *like* the load-dependent, class-based queue. As previously mentioned, considering the total amount of requests that reaches the queue, we are able to estimate the probability p_R of receiving a read request, and the corresponding $p_W = 1 - p_R$ for the writes. The average T_{mem} for the queue will be

$$T_{mem} = p_W * T^W_{mem} + p_R * T^R_{mem} \tag{5.9}$$

because of the load-dependent behavior, we apply the formula to the service time of each queue size (we indicate with $T^{(k)}_{mem}$ the service time with queue size of k):

$$\begin{aligned} T^{(0)}_{mem} &= p_W * T^{W(0)}_{mem} + p_R * T^{R(0)}_{mem} \\ T^{(1)}_{mem} &= p_W * T^{W(1)}_{mem} + p_R * T^{R(1)}_{mem} \\ \ldots &= \ldots \\ T^{(128)}_{mem} &= p_W * T^{W(128)}_{mem} + p_R * T^{R(128)}_{mem} \end{aligned}$$



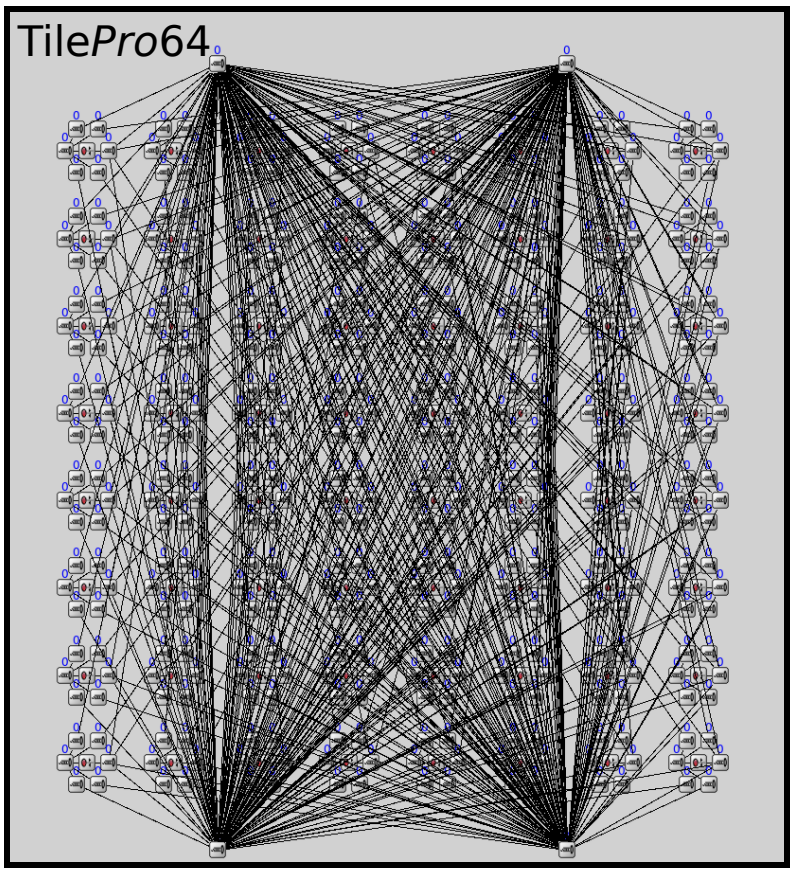


Figure 5.27: Architectural model of the Tile*Pro*64 processor: conceptual representation and EQNSim implementation.

To summarize, we present in Tables 5.14 and 5.15 the parameters of the model. We divided them in "architecture" parameters, that are fixed with the values already gathered, and "program" parameters, that need to be selected according to the program that is executed on the architecture.

Parameter	Description	Multiplicity
L_{mem}	Memory latency, composed of $T_{mem} + T_{mc}$	One per memory
T_{mc}	Fixed time to prepare a memory response	One per memory
T_{mem}	Service time of the memory, defined as $p_R * T^R_{mem} + p_W * T^W_{mem}$	Many per memory
T^R_{mem}	Service times of the memory for read requests.	Many per memory
T^W_{mem}	Service times of the memory for write requests.	Many per memory
L^1_{net} L^2_{net} L^3_{net} L^4_{net}	Network latency to reach the memory controllers	One per core

Table 5.14: Summary of the model parameters that depend on architecture.

Parameter	Description	Multiplicity
n	Number of processors that generate memory requests	One
T_p	Computing time between two memory requests. Depends on p^l_{miss} and p^r_{miss}	One per core
p_{mod}	Probability of replacing a modified cache line	One per core
p_{mc1} p_{mc2} p_{mc3} p_{mc4}	Probability of sending a memory request towards the specified memory interface	One per core
k	Corrective factor to take into account the exact moment in which the processor enter and exit the stall (required to correctly estimate the memory latency)	One per core
p_W	Probability of receiving a memory write request	One per memory

Table 5.15: Summary of the model parameters that depend on the program.

5.7.1 Evaluation of R_q for store_linear

We start evaluating the model over the previous benchmark, **store_linear**. We selected the values of the model according to the benchmark, in particular:

- $p^l_{miss} = 1$: each iteration of the benchmark will produce a load request that result in a miss, because of the offset chosen between elements (see Listing 5.3).
- $p^s_{miss} = 0$: store are executed on previously loaded addresses, so the corresponding line will be already in cache.
- $p_{mod} = 1$: we write each word we are loading, so at steady state each line allocated on the cache will be modified, producing a write request when replaced.
- $p_W = 0.5$: given the previous vale of p_{mod}, we have that each read is coupled with a write, so that half of the requests reaching the memory interface will represent *write requests*.
- $p_{mc1} = 1$, as all the requests are directed towards the first memory controller of the chip; obviously, the other probabilities are set to 0.
- T_p, on the other hand, is gathered by the profiling of the code and set with the values already presented.

In Figure 5.28 we report the results for three values of T_p that covers a highly loaded system, a moderately loaded one and finally a lightly, practically not loaded, system. We can definitely say that we are able to correctly predict the qualitative behavior of the response time of the memory, yet we already observe from the graph numerical differences between the predicted and the real R_q. These values are presented in Table 5.16. With the first values (up to 4 cores) we have, as expected, an important difference because we did not model the possible page hits at the memory controller level. Overall, the maximum error is always within 20% in all our tests, while the average error stops at 6.16%, 9.03% and 8.29% for the analyzed T_p values of 4815 975 and 40 clock cycles, respectively. We think the model approximation is acceptable for our purpose.

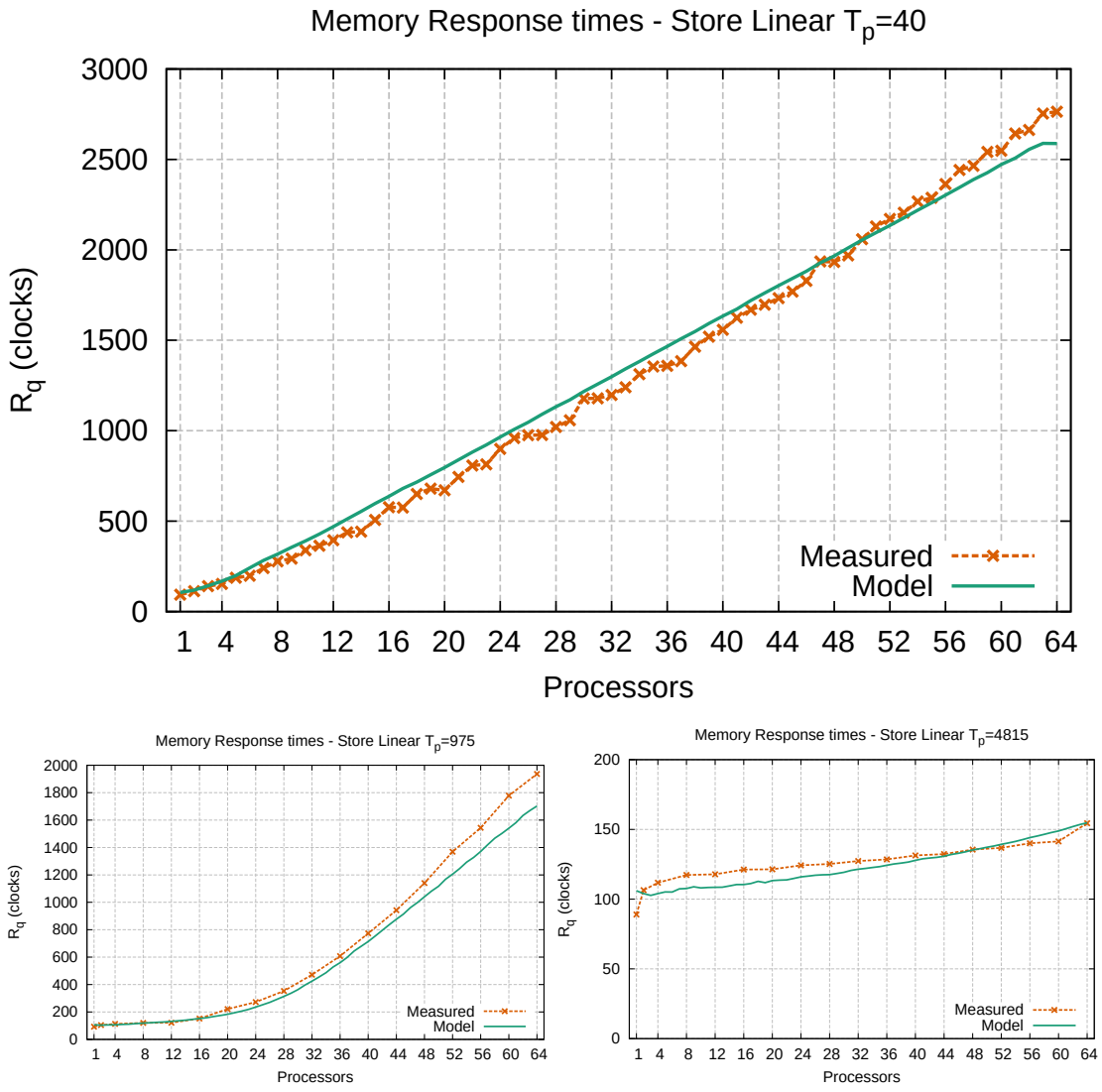


Figure 5.28: Comparison between architecture and model values of R_q for the **store_linear** benchmark. Times in clock cycles.

	$T_p = 4815$				$T_p = 975$				$T_p = 40$			
	R_q		Error		R_q		Error		R_q		Error	
	Real	Model	Abs	%	Real	Model	Abs	%	Real	Model	Abs	%
1	88,96	106,00	17,04	19,15	92,50	106,00	13,50	14,60	93,84	106,00	12,16	12,95
2	106,46	103,98	2,48	2,33	104,35	103,59	0,76	0,73	113,45	118,24	4,79	4,22
4	111,78	103,98	7,80	6,98	113,12	106,33	6,78	6,00	153,07	168,65	15,57	10,17
8	117,30	107,64	9,66	8,23	120,24	118,57	1,66	1,38	278,04	318,05	40,01	14,39
12	117,84	108,47	9,36	7,94	121,79	131,51	9,72	7,98	394,12	469,84	75,72	19,21
16	121,21	110,45	10,77	8,88	152,03	152,04	0,01	0,01	576,82	637,32	60,49	10,49
20	121,38	113,30	8,08	6,66	221,36	183,10	38,26	17,28	670,63	796,09	125,46	18,71
24	124,24	115,94	8,29	6,68	272,46	236,07	36,40	13,36	899,40	965,57	66,17	7,36
28	125,23	117,59	7,64	6,10	353,47	313,88	39,59	11,20	1020,45	1132,68	112,22	11,00
32	127,33	121,48	5,84	4,59	471,64	424,65	46,99	9,96	1197,92	1298,56	100,64	8,40
36	128,47	124,31	4,16	3,24	607,55	559,23	48,32	7,95	1358,80	1466,50	107,70	7,93
40	131,34	127,84	3,50	2,67	774,00	713,68	60,32	7,79	1557,50	1633,55	76,05	4,88
44	132,30	130,80	1,50	1,13	942,24	876,82	65,42	6,94	1732,73	1802,64	69,90	4,03
48	135,50	135,30	0,21	0,15	1140,19	1040,19	100,00	8,77	1932,66	1966,56	33,90	1,75
52	136,74	139,33	2,59	1,89	1368,66	1205,40	163,25	11,93	2170,76	2134,55	36,21	1,67
56	140,02	144,14	4,12	2,94	1543,84	1370,52	173,32	11,23	2363,72	2302,31	61,40	2,60
60	141,42	148,90	7,47	5,29	1778,14	1540,91	237,23	13,34	2548,40	2471,83	76,57	3,00
64	154,43	154,76	0,33	0,22	1936,49	1702,47	234,02	12,08	2763,72	2587,79	175,94	6,37

Table 5.16: Comparison between architecture and model values of R_q for the **store_linear** benchmark. Times in clock cycles.

5.7.2 Evaluation of R_q for store_linear with a different store rate

One of the most important part of the queueing network model is the memory queue. In particular, we wanted to test its behavior under a different mix of read/write operations, to check if the gathered values for $T_{mem}^{(x)}$ hold with different streams of requests. To obtain this we modified the **store_linear** benchmark, in such a way that the code is basically unchanged, but we do perform the store operation once every two loads. In fact, we added a parameter to the benchmark (that we called S_p) to determine the probability of issuing a store after the load. The previously studied run have a $S_p = 1$, while this new run have $S_p = 0.5$. This way we have that, on steady state, only *half* of the cache lines are in a modified state. A direct consequence is that we have a different p_{mod}, that in turn affect p_W. We report in Table 5.17 the parameters of this benchmark, compared with the previous one. The resulting R_q are depicted in Figure 5.29, while numerical values can be found on Table 5.18. The results are aligned to the previous ones, with only slightly higher errors; this is a very good sign as we sensibly changed the behavior of the program (w.r.t the kind and frequency memory requests), and we were still able to capture the qualitative behavior with a quite good accuracy, and an average error of 10.74%.

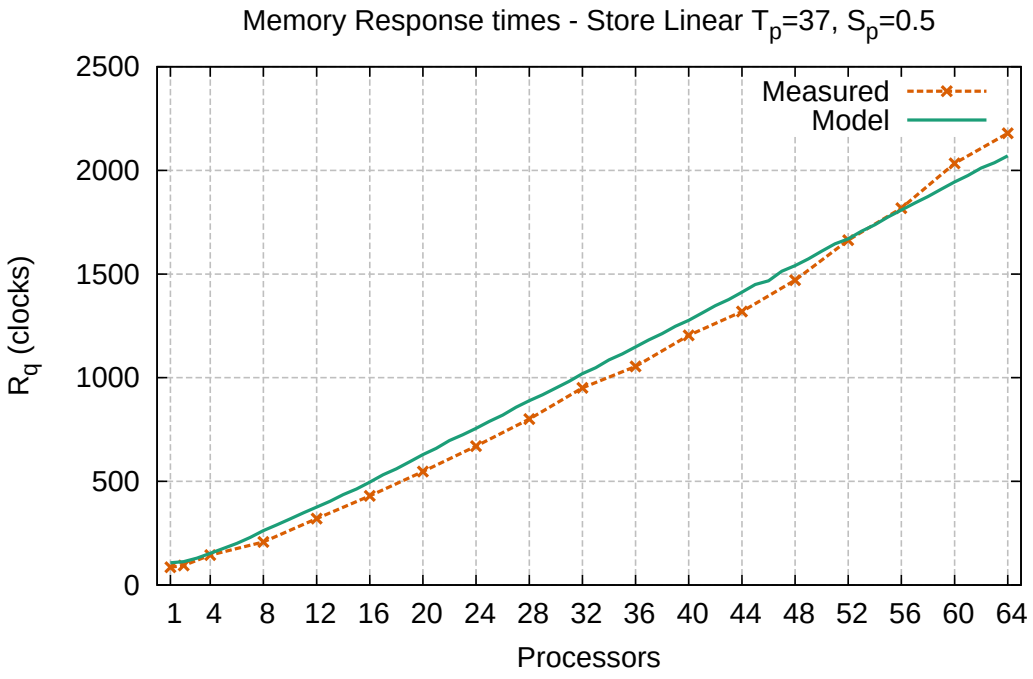


Figure 5.29: Comparison between architecture and model values of R_q for the **store_linear** benchmark with $T_p = 37, S_p = 0.5$. Times in clock cycles.

Store_linear		
	$S_p = 1$	$S_p = 0.5$
T_p	40	37
p^l_{miss}	1	1
p^s_{miss}	0	0
p_{mod}	1	0.5
p_W	$\frac{1}{2}$	$\frac{1}{3}$
p_{mc1}	1	1

Table 5.17: Difference in parameters between the two **store_linear** runs.

	R_q		Error	
	Real	Model	Abs	%
1	85,31	107,00	21,69	25,42
2	94,19	112,69	18,50	19,64
4	144,67	153,09	8,41	5,82
8	207,97	262,16	54,19	26,06
12	320,08	375,94	55,86	17,45
16	429,80	496,67	66,88	15,56
20	547,00	629,41	82,41	15,07
24	670,12	755,30	85,18	12,71
28	800,07	889,45	89,38	11,17
32	950,91	1019,94	69,03	7,26
36	1054,25	1148,59	94,35	8,95
40	1204,12	1276,99	72,87	6,05
44	1318,75	1412,15	93,39	7,08
48	1470,23	1540,69	70,46	4,79
52	1663,55	1669,82	6,27	0,38
58	1817,59	1808,17	9,42	0,52
60	2034,16	1944,09	90,07	4,43
64	2179,11	2069,90	109,21	5,01

Table 5.18: R_q values for the **store_linear** benchmark with $S_p = 0.5$.

5.7.3 Considerations on the accuracy of the model

We tested the model with two different benchmark, obtaining approximations within $\sim 10\%$ of average error on the values of R_q. We believe that this is a remarkable result, that allow us to use this model to estimate the behavior of a parallel program executed on the Tile*Pro*64 architecture. Of course, we could try to refine the model, and in particular its parameters, to obtain even better approximations. However, we believe we reached a good model, and can move on the *application* of it in parallel programs.

It is important to note that we did study the model by means of few, simple benchmarks. This was intended to put us in very simple and clear situations, to easily determine the model parameters, and focus only on the definition of the queueing network. However, at this point the model already seems very "stable", and able to capture the wide range of parallel programs: the examples already prove that the model correctly mimic the behavior of the real architectures.

The main problem is to correctly estimate T_{mem}, that seems to depend on the single program behavior. However, the idea of gathering T^R_{mem} and T^W_{mem} separately

seems effective, as we used the gathered values on another program, with a different amount of read and write operations (and thus a different T_{mem}) with very good results. We also saw that the characteristics of the streams of requests, and thus the possibility of exploiting features like the open page policy, have a minimal impact on a highly parallel architecture, because all the streams mix together at the memory interface, and thus lose all their locality properties.

For the sake of conciseness, we will not report further studies on different applications here: we will use the model throughout the following chapters, on different programs, as a further prove of its efficacy.

5.8 Summary

With this chapter we studied the Tile*Pro*64 architecture in order to develop a queueing network model to evaluate the effect of contention on the shared memory system. We started from the model proposed by Bhandarkar[36], but ended with something much different. It is notable to notice that our queueing network does not resemble any of the models available in literature and presented in the previous chapter. This is due to several aspects, some of which probably related to the single-chip nature of this multiprocessor, and to the increasing complexity in processor and memory technology. However, the model is able to approximate the response time of the loaded system with an average error within $\sim 10\%$. Given the complexity of the architecture, we considered this a very good results, that will allow us to compare different parallel programs with sufficient accuracy.

Yet, some simplifications had to be done to reach this level of accuracy. In particular, we assumed that the program works with incoherent memory areas, because of the complexity of modeling the automatic cache coherence mechanisms of the Tile*Pro*64 architecture. At this point we are not able to tell if this will strongly limit the performance of the architecture, although some works already presented in the previous chapters ([1] and [104]) suggest that we can manually handle cache coherence without a lot of overhead. Chapter 7 reports more experiments, on the Tile*Pro*64 architecture, to further confirm the feasibility of using software-based cache coherence in conjunction with parallel patterns, to obtain good performances, in some cases even better w.r.t. hardware-based approaches. Even so, the possibility of losing a bit of performance is still is in line with our methodology, and in particular with the idea of only selecting *predictable* implementations in order to compare them.

Finally, with this chapter we also presented **EQNSim**, the simulation framework we used to rapidly test and compare modeling ideas based on concepts of queueing networks.

Part III

Optimizations

Chapter 6

Exploiting Multiple Memory Controllers

Historically, one of the most important characteristic of a parallel architecture was its "memory configuration": the bandwidth required by a parallel machine cannot be obtained by a single memory, so multiple interfaces are adopted; in this scenario:

- Each processor may be able to natively access only a single portion of the global memory; in this case we talk of **distributed memory** architectures, such as in clusters.
- Each processor may be able to natively access the whole global memory; in this case we talk of **shared memory** architectures, such as in current multi-cores.

In this thesis we are, of course, interested in the second class of architectures. Shared Memory Architectures are divided in two sub-classes:

- **Uniform Memory Architecture** (**UMA**), where the whole memory is used as a *centralized entity*, where all the requests are served. In this case, the *access time* to the memory is uniform (i.e. it is the same for each memory address, for all the processors).
- **Non-Uniform Memory Architecture** (**NUMA**), in which the whole memory is *sliced* in parts, each one working independently, in a way that each slice offer different *access time*.

The choice between uniform and non-uniform memory architectures is usually driven by the *size* of the parallel architecture; for small-size parallel machines simple interconnection networks (i.e. buses) between the processors and the memories favors a uniform memory architecture. As the number of processors grows however, complex, not-centralized networks are required to effectively connect the large number of processors; in this case a uniform access to the memory is no longer feasible, so NUMA configurations are used. Aside from technical limitations, the

two classes offers different performances (even in comparable configurations) and requires different programming methodologies.

UMA architectures, because of the uniformity of access times, favor a programming model in which processes are anonymous (i.e. they can be executed on any of the processors) and indiscriminately access each location of the memory. Parallel programming models such as OpenMP are quite effective on these machines. These architectures are also usually called Symmetric Multiprocessing (SMP).

In **NUMA** architectures, of course, the varying distance from the memory requires a more careful implementation, in which each process (running on a specific processor) *should* favor the use of the *nearest* memory, and limit the traffic towards other memories just to exchange data with other processes. In fact, NUMA-oriented applications exploit the same locality concepts required in distributed memory architectures, resulting in the fact that NUMA-oriented shared-memory programs often employs the very same parallelization techniques of their corresponding message passing programs[60] and, in several cases, they actually use message-passing models implemented on top of the shared memory[60, 95, 78, 147] for portability and performance reasons.

Original NUMA followed a plain architecture, composed of a variable number of nodes, each containing a processor, a memory and a network connection. In time, however, *clustered* architectures emerged both in the research and in the commercial world. In a clustered NUMA each node is composed by several processors connected to the node memory in UMA configuration, while nodes are connected together to form a large NUMA machine. Commercial examples of clustered NUMA were the SGI Altix 3700 (two Itanium processors per node, up to 256 nodes)[75] and the Cray X1 (four MSP processors per node, up to 1024 nodes)[74]. Conceptually this is, of course, still a NUMA architecture because of the different distances of the off-node memories. However, the existence of a SMP configuration inside each node theoretically allowed mixed UMA-NUMA parallel programming models. In fact this practice has never been really used because of the very limited amount of parallelism inside each node (2 and 4 in the previous architectures) w.r.t to the total number of processors (hundreds to thousands), so the whole architecture was normally used as a flat NUMA.

The multi-core appearance significantly changed the scenario: plain NUMA architectures no longer exist, given the multiple cores per chip, and very-large NUMA solutions (more than 256 cores/processors) were slowly substituted by distributed memory solutions (as of today one of the few[1] large-scale NUMA is the SGI UV, direct evolution of the Altix series, that supports two 8-core Xeon processors per node and up to 128 nodes).

However, NUMA architecture are today more popular than ever: the increasing memory bandwidth requirements of multi-cores drove to the inclusion of the memory

[1]Probably the only one still commercially available

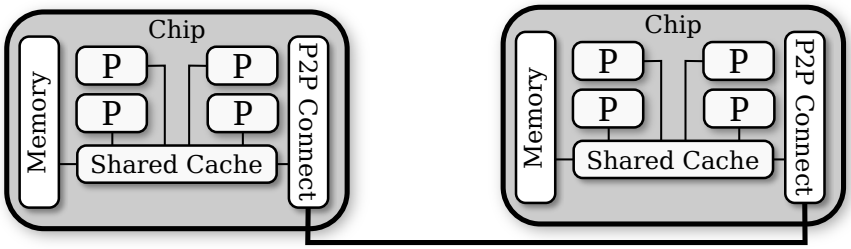


Figure 6.1: Generic architecture of a multi-chip machine.

controller *inside* the processor chip, and later even to the increase of the number of controllers per chip: the Power7 and the Tile*Pro*64 both contain four memory interfaces on-chip. This evolution trend, if confirmed, will result in a Non-Uniform memory architecture *on-chip*, where each core will be physically nearer to one of the (many) memory controllers. On top of this, the classic use of multiple chips in a node, that in the past determined uniform architectures, is now producing non uniform architectures, as each core will be obviously nearer to the slice of memory controlled by its on-chip interface. Notable examples of machines that exploit this model are the HP Integrity Superdome 2 that connects 32 Itanium chips, each one with 8 cores and two memory controllers[145], the IBM Power795 that connects 32 Power7 chips, each one with 8 cores and four memory controllers[102], but also small servers like HP ProLiant DL987 (8 chips, up to 10 cores per chip) or the HP ProLiant DL360e (2 chips, up to 8 cores per chip). The generic architecture of all these machines is depicted in Fig 6.1: on-chip processors shares the memory controller(s) and *possibly* a cache level, and are connected to the other chips by using a set of point-to-point connections.

6.1 Programming multi-cores

In this scenario we can definitely say that basically every parallel architecture implement a non uniform memory. In this thesis we are interested in "small" clustered NUMAs, with 2 to 8 nodes and from 6 to 16 (currently) cores per node, in which nodes are tightly coupled, usually exploiting crossbar interconnections and on-board wiring, making the access times of "remote" memories not much higher w.r.t the "local" ones. Table 6.1 shows the difference in remote access of the SGI UV w.r.t the target architectures selected for this thesis: in the first case we have a remote latency up to 15 times the local one; in the others, less than 2 times.

Finally, we have the Tile*Pro*64 processor that gives an insight of a possible future, when the number of cores per chip will be so high that the non-uniform memory access will be present even inside the chip. For simplicity reasons in Table 6.1 we considered 4x4 squares of cores as a NUMA node, because of their common *nearest*

Machine	Processors		Latency	
	Cores/node	Nodes	Local Memory	Remote Memory
SGI UV	8	256	65ns	up to 1000ns
Opteron 6176	6	5	65ns	120ns
Xeon E7-4820	8	4	130ns	193ns
Xeon E5-2650	8	2	90ns	140ns
Tile*Pro*64	16	4	[76, 91]ns	[93, 106]ns

Table 6.1: Local and Remote latencies of large, on-node and on-chip NUMA[131, 132, 150].

memory; nevertheless in this architecture *each* core of the "node" have a different memory latency (for this reason we report latency intervals). In this architecture the latency difference between local and remote access is even smaller.

With these access times, is not really clear how much the locality concepts usually required to exploit NUMA architectures are necessary to design parallel applications. To give an idea, we consider the performance model presented in Chapter 4, Section 4.2.5.1:

$$T_{w-calc} = Fixed\_Time + L_n\_misses \times Predicted\_Memory\_Latency \quad (6.1)$$

Let us say that the memory is not a bottleneck, so $Predicted\_Memory\_Latency$ is equal to the values in Table 6.1, and that an half of the memory requests goes to the local memory, while the other half is directed towards the remote memory. As reported in Table 6.2, the impact of the remote requests result in an increase of the memory-related part of the performance model that goes from 800% in the SGI UV to a 19% in the Tile*Pro*64. Of course this performance degradation will be further mitigated, depending on the values of $Fixed\_Time$ in Eqn. 6.1. However, if the application is "memory bound", i.e. its performance is limited by the memory latency, maintaining only local accesses should offer improvements up to $20 - 30\%$ on the two multicore-based architectures.

Nevertheless, a very common approach today is to design (or reuse) SMP-oriented parallel applications for multi-core architectures. Despite the possible performance improvement we exemplified, no scientific studies exists to quantify the real improvement in benchmarks or real-life applications. With this chapter we start addressing the problem, by studying the impact of the Unified/Non-unified programming model on our set of target architectures, by mean of specific benchmarks. We address two different, yet strictly correlated, allocation concepts: **Memory Allocation**, i.e. how and where allocate memory for the parallel application, and **Process Allocation**, i.e. how to place the processes/threads of the application on the parallel architecture.

Machine	Memory-related part of the performance model
Local requests only	$L_n\_misses \times Local\_Latency$
SGI UV	$\frac{L_n\_misses}{2} \times Local\_Latency + \frac{L_n\_misses}{2} \times Remote\_Latency$
	$= L_n\_misses/2 \times Local\_Latency$
	$+15.38 \times L_n\_misses/2 \times Local\_Latency$
	$= 8.19 \times L_n\_misses \times Local\_Latency$
HP ProLiant DL360e	$\frac{L_n\_misses}{2} \times Local\_Latency + \frac{L_n\_misses}{2} \times Remote\_Latency$
	$= L_n\_misses/2 \times Local\_Latency$
	$+1.63 \times L_n\_misses/2 \times Local\_Latency$
	$= 1.31 \times L_n\_misses \times Local\_Latency$
Tile*Pro*64	$\frac{L_n\_misses}{2} \times Local\_Latency + \frac{L_n\_misses}{2} \times Remote\_Latency$
	$= L_n\_misses/2 \times Local\_Latency$
	$+1.39 \times L_n\_misses/2 \times Local\_Latency$
	$= 1.19 \times L_n\_misses \times Local\_Latency$

Table 6.2: Performance of a parallel program executed using only local memory or local and remote memory, on three different NUMA architectures.

6.1.1 Memory allocation models

We start by considering the problem of memory allocation, in presence of multiple possibilities (i.e. multiple memory interfaces). With the introduction of multiple interfaces on a single chip, the research community started to tackle the problem at the operating system level.

The Operating System is responsible, among other things, of handling the scheduling of processes and the virtual-physical memory mapping, so that - from a certain point of view - it can be considered the right place to address this problem. In this area several work stem from the original page allocation and migration ideas for NUMA architectures. The common allocation policy currently used on most operating systems in NUMA architectures is the first-touch allocation policy, where memory is allocated on the nearest interface of the processor that generate a page fault exception (i.e. the first processor that "touch" the page). This way, if the process running on that processor is the only one using the page, *and* the process is not moved onto another processor during its execution, the O.S. is able to offer the best allocation. However, processes may be moved, and memory can be shared. In both cases the first-touch provides non-optimal results. Several works addressed the problem, by proposing profile-guided page allocation for parallel programs[119, 120] (i.e. the program is initially executed, augmented with profiling code, to obtain an approximate memory access trace, that is used to select a pseudo-optimal allocation for the specific program), or by proposing page migration techniques[117, 159, 175] (where pages are migrated from a memory interface to another, trying to obtain a trade-off between the remote access latency and the migration overhead).

The first paper that address the problem *specifically* for multi-cores is [16], where the authors propose:

- An improved first-touch policy called "Adaptive First-Touch Page Placement Policy", where memory allocation still occur on the "first-touch" event, but the destination interface is selected by using an objective cost function that do not consider only the memory distance, but also other factors such as the average queueing delay at each interface.

- A "Dynamic Page Migration Policy" that moves pages to keep the load balanced among the memory interfaces.

Similar ideas have been then introduced in other recent works[70, 156]. However, all these works assume that the program has been already designed and compiled; it does not even need to be parallel: from the o.s. point of view, we just have a set of processes and threads running concurrently; they may be parts of a parallel application, or just a set of programs executed in a multiprogrammed environment.

In this thesis we work with a very different perspective: *the program is yet to be compiled/designed*, and we want to select the choice that should be, with reasonable assurance, the best solution.

In this process we can also count on a deep knowledge of the parallel algorithm implemented: because of the use of parallel patterns, we can assume that we know, for each process/thread of the parallel program, which memory areas are accessed locally. In this scenario, we study the two different types of memory allocations commonly used on SMP-oriented and NUMA-oriented parallel applications.

6.1.1.1 SMP-like memory allocation

This is considered the current state-of-the-art in multi-core parallel programming: the machine is abstracted as a Uniform Memory Architecture, in which locality is not required to obtain good performance results.

The main limitation of this approach is that, because of the underlying NUMA organizations, the physical address space is "sliced" among the memory interfaces, instead of interleaved as in classic UMA architectures. This difference (shown in Fig 6.2a and 6.2b) makes it more difficult to exploit the aggregate bandwidth of the whole memory, and depend on the "first-touch" memory allocation of the O.S. Interleaving can be still be implemented, but at the "virtual memory page" level: the programmer (or the operating system) can select the physical address of each page so that contiguous virtual memory pages resides in different memory interfaces, as depicted in Fig 6.2c. Of course the two interleaving approaches are not equivalent, as in the second case the amount of interleaving heavily depends on the memory access pattern of the parallel application.

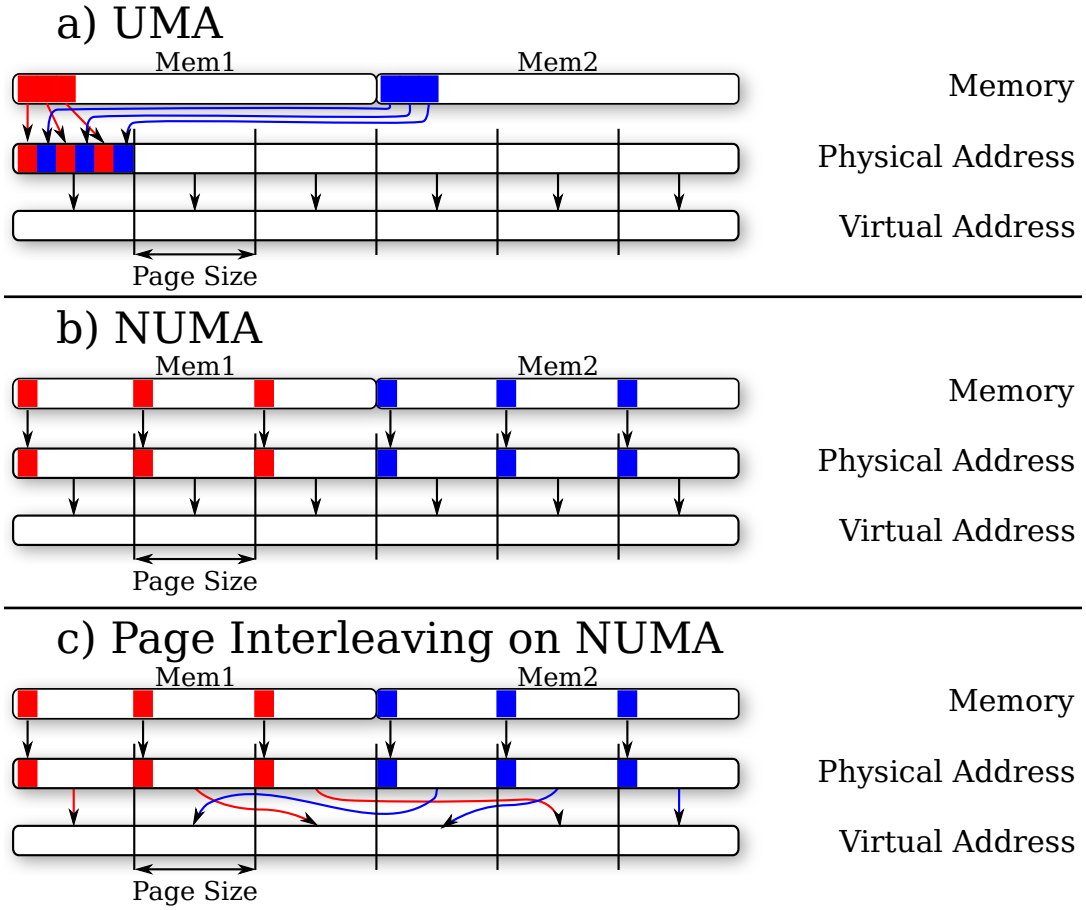


Figure 6.2: Different memory allocation policies.

6.1.1.2 NUMA-like memory allocation

A different approach is to consider these architectures as common NUMA machines, writing parallel programs that exploit the concept of memory locality, in which each process allocate its working memory on the nearest memory interface. Given the underlying non uniform architecture, we expect this solution to provide a performance level better or, at least, equal w.r.t the SMP-like allocation; surprisingly, this approach has not yet been studied and compared with the previous one, so we are not (yet) able to quantify the real advantage of this allocation model.

6.1.2 Process allocation

As previously mentioned, in a pure SMP architecture processors are *anonymous*, i.e. each process[2] can run indiscriminately on any processor. Nevertheless, in HPC

[2]As in most part of the thesis, we refer to process and thread as synonyms: for the real implementation, we select one of them depending on the possible performance improvement

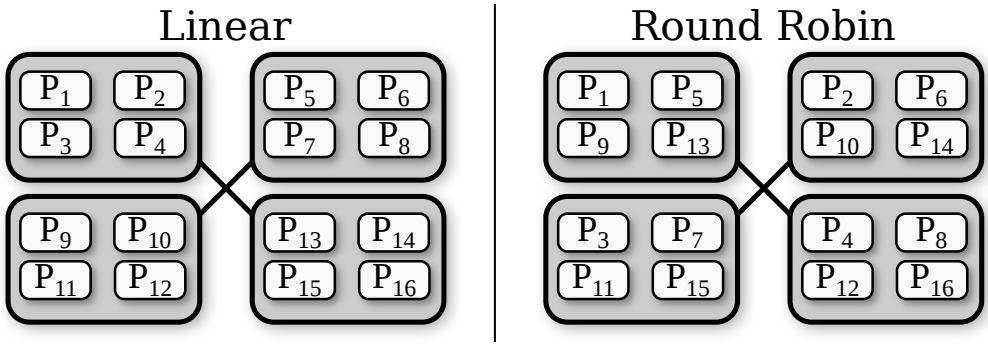


Figure 6.3: Different process allocation policies.

programs, to favor cache reuse, a common practice is still to *fix* each process/thread on a specific processor, so that the O.S. scheduler is guaranteed to not modify the mapping during the execution. In the presence of a parallel program executed on a clustered NUMA, it often happen that we do not exploit the overall number of cores, but we limit the execution to a smaller set of cores. In this case, it is important to determine if it is more efficient to try to completely fill single clusters of cores (i.e. fill the first node, then fill the second, and so on) or to *balance* the utilization among clusters (i.e. allocate a core per node, then two cores per node, and so on).

The process allocation policy may affect the performance of parallel programs because of the shared level of cache present in many multi-core chips: sharing a cache may reduce the performance if the cache working set of all the processes/threads do not fit the single, shared cache; but may also increase the performance in case of frequent data exchange among the processes/threads, as it can better exploit automatic cache coherence.

Process allocation also affect the memory performance, and play an important role when mixed with the memory allocation policies, because of the different placement of processes w.r.t memories. For this reason we will study two different process allocation policies:

- **Linear**, in which we try to fill one NUMA cluster (i.e. chip) at a time, thus increasing the amount of cache sharing, by allocating processes linearly on the cores (i.e. with 5 processes we uses cores from 1 to 5, and so on).

- **Round-Robin**, in which we try to balance the amount of nodes occupied in each cluster, thus decreasing the amount of cache per process/thread.

6.2 Evaluation by mean of synthetic benchmarks

We start the study of our process and memory allocation policies with a synthetic benchmark, in which we are able to determine the memory request/computation

ratio. This way we can easily determine and test the conditions that makes the memory a bottleneck, and focus our study on those cases.

We developed our synthetic benchmark starting from the code in Listing 6.1. The code, reported in MIPS-like assembler, consist in a simple loop where:

1. we read an element from an array;
2. we add the element to an accumulator;
3. we increase the address pointer (by 64, assuming 64-byte cache lines);
4. we increase the element counter by 1;
5. we check if we read the whole array, otherwise return to the beginning.

This fairly simple code will probably be enough to stress the memory, as we perform a memory read every 6 instructions. To control the amount of concurrent memory requests, we need to put the **mfence** instruction to stall the processor until the load is not completed. While this is not needed for the add (i.e. the add requires the result of the load), it may happen, with highly superscalar and out-of-order processors, that multiple iterations of the loop are executed concurrently (i.e. while waiting for instr. 1 to complete, the array indexes are updated and the processor begin with the next iteration). The **mfence** instruction ensure that we wait the completion of the load before continuing the execution.

From the code in Listing 6.1 we can easily derive a benchmark in which multiple loads and/or multiple adds are executed, allowing us to perform multiple loads concurrently (but in a controlled environment, in which we specify their number), and to increase the amount of processor calculation between memory loads. The result is exemplified in Listing 6.2, with two loads and two adds, and in Listing 6.3 in a generic way.

For portability reasons, the final benchmark is written in C code, so that we do not need to re-write it for each architecture; of course the compiled code is basically the same of the one in Listing 6.3.

To simulate a parallel program, the code will be executed in multiple copies, using threads. We will allocate up to one thread per core, because some architectures do not support hardware multithreading. Each thread works on a different array, so that we are able to test the various memory allocation policies on the array (that represent the *local* data of each thread on a parallel application). All the threads synchronize at the beginning of the execution by means of a pthread barrier, so that we guarantee that they all start working on their arrays more or less at the same time. The code is executed 100 times to average the results.

Given the possible correlation between memory and process allocation, we studied all the combinations:

```
LOOP:    lw $6,$7 #R7: Array
    address
         mfence
         add $4, $4, $6
         addi $7,$7,64
         addi $1,$1,1 #R1:
    Array index
         bne $1, $0, LOOP #R0:
    Array size
```

Listing 6.1: Base Synthetic Benchmark.

```
LOOP: lw $6,$7
  lw $8,$9
  mfence
  add $4, $4, $5
  add $4, $4, $5
  addi $7,$7,64
  addi $9,$9,64
  addi $1,$1,1
  bne $1, $0, LOOP
```

Listing 6.2: Multi load-add benchmark.

```
LOOP: lw $6,$7 #R7: Array1 address
  # Sequence of Load
  mfence
  add $4, $4, $5
  # Sequence of add
  addi $7,$7,64
  # Sequence of Address Update
  addi $1,$1,1 #R1: Array index
  bne $1, $0, LOOP #R0: Array size
```

Listing 6.3: Generic Skeleton of the Synthetic Benchmark.

- **SMP-like, linear**: Memory allocated in an SMP-like way, interleaving virtual pages among the controllers; processes allocated in a linear way, using all the cores of a chip before using the other chips.

- **SMP-like, RR**: Memory allocated in an SMP-like way, interleaving virtual pages among the controllers; processes allocated in a Round-Robin way, balancing the used cores among the chips.

- **NUMA-like, linear**: Memory allocated in an NUMA-like way, allocating, for each thread, the local data on the nearest controller; processes allocated in a linear way, using all the cores of a chip before using the other chips.

- **NUMA-like, RR**: Memory allocated in an NUMA-like way, allocating, for each thread, the local data on the nearest controller; processes allocated in a Round-Robin way, balancing the used cores among the chips.

These combinations will also be compared w.r.t using a single memory interface and a linear process allocation. While this is inserted just a baseline, to compare the gain obtained by using multiple memory controllers, it should be noted that in many cases parallel applications not specifically designed for this kind of systems fall in this category.

6.2.1 Experimental results on the target architectures

Here we report the results obtained by running the benchmark on the four target architectures listed in Chapter 3, Section 3.7:

- two Intel Xeon® E5-2650, Sandy-Bridge based architectures, composed of 8 cores, 20 MB of shared cache and a memory controller per chip, running at 2.00GHz; we will refer to this machine as **SandyBridge**.
- four Intel Xeon® E7-4820, Westmere-EX based architectures, composed of 8 cores, 18MB of shared cache and a memory controller per chip, running at 2.00GHz; we will refer to this machine as **Nehalem**.
- two AMD Opteron™ 6176, for a total of 4 Magny-Cours based chips, composed of 6 cores, 6MB of shared cache and a memory controller per chip, running at 2.3GHz; we will refer to this machine as **AMD**.
- a single Tilera TILEPro64™, composed of 64 cores, no shared cache but 4 memory controllers, running at 866MHz, that will be called **Tilera**.

The first three architectures are all connected by point-to-point connections among chips, forming crossbar-like interconnection. All the remote memory interfaces are therefore equidistant w.r.t each processor. In the Tile*Pro*64 machine things are slightly different, mainly because of on-chip NUMA: we have a mesh so *each* interface have a different distance, but remote latencies are only 20% higher than the local ones. In the Tile*Pro*64 implementation we also use non-coherent memory spaces for the local arrays, as this allow to avoid the DDC mechanism and is not a problem for the benchmark semantics because each array is local to a thread.

1 Load, 1 Add per iteration

We start with the simple case of a load and an add per cycle; from this, we will find if we need to further increase the memory pressure, by adding multiple loads, or decrease it, by adding multiple adds. Execution times per iteration are depicted in Figure 6.4, in clock cycles. Given the parameters of the benchmark (1 Load, 1 Add per iteration), we can consider the result an accurate evaluation of the average memory response time (i.e. the execution time of each iteration is dominated by the time the processor wait for the load response). The graphs show the behavior as the number of cores executing the benchmark is increased. Each line represent one of the combinations previously presented. **Single{1-4}-linear**, on the other hand, represent a configuration that uses a single memory interface. The first interface (corresponding to **Single1-linear**) belongs to the first chip used, so that it can be considered a "local" memory for the initial parallelism degrees while, on the other hand, interfaces 2 to 4 represent the "remote" memories.

We expect an almost constant time as long as the memory is not a bottleneck; then, a large increase in the times when the memory becomes the bottleneck of the

system, as its R_q increases. We show the result ordered per maximum number of cores, starting with the **SandyBridge** architecture, up to the **Tilera**. Surprisingly, results are very different, depending on the architecture.

The first row shows two architectures in which the memory is not a bottleneck with this benchmark. This is a notable result, as the benchmark code is indeed very "fine-grained", in the sense that the amount of register-register operations between two loads is very small. Yet the memory subsystem is able to handle 16 and 24 cores, respectively, with almost no degradation even with a single memory interface.

On the other hand, the second row shows a completely different behavior, where a single memory controller is indeed not sufficient to handle all the processors and the use of multiple controllers, regardless of the policy, offer significant improvements on the R_q, and therefore the execution time.

While these results are interesting per-se, it is quite difficult to understand how this increase in the execution time affect the performance of a parallel program. We therefore used a simple yet effective model to estimate the hypothetical scalability of a parallel application designed on top of this synthetic benchmark.

We define our application as a program that execute the measured code for a certain amount of iterations. Its completion time is determined as

$$T_c^{seq} = T_{it}^{seq} * N_{it} \tag{6.2}$$

while the completion time of a parallel version with n threads that follows a *map paradigm* will be

$$T_c^n = T_{it}^n * \frac{N_{it}}{n} \tag{6.3}$$

where each thread execute a fraction of the total iterations, with its own iteration time. From this we can calculate a hypothetical speed-up, w.r.t the sequential code:

$$S^n = \frac{T_c^{seq}}{T_c^n} = \frac{T_{it}^{seq} * N_{it} * n}{T_{it}^n * N_{it}} = \frac{T_{it}^{seq} * n}{T_{it}^n} \tag{6.4}$$

The corresponding graph of S^n is depicted in Figure 6.5, taking the time with 1 core of **Single1** as T_{it}^{seq}. The results show that, when the memory is not a bottleneck, the Speed-up of the hypothetical application is good, usually aligned with the theoretical one. When, on the other hand, the memory is a bottleneck, we assist to a behavior in which the speed-up grows up to a point, and then remain still. Depending on the architecture, the overall scalability can be pretty bad, especially when using a single interface.

SandyBridge A detailed analysis of this architecture show several important behaviors. This machine has two memory controllers, so we have only **Single1** and **Single2**. Yet their performance difference require some considerations: with a single thread, using the first or the second memory controller gives very different response

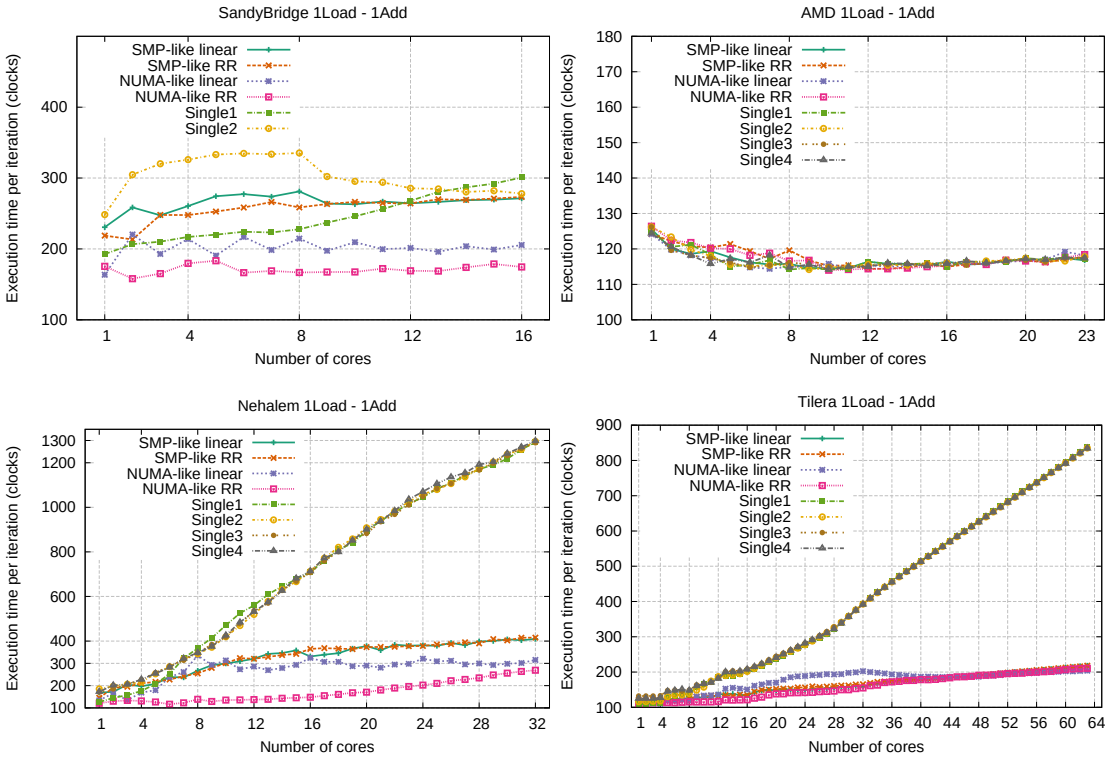


Figure 6.4: Average execution times per iteration, 1Load-1Add per iter.

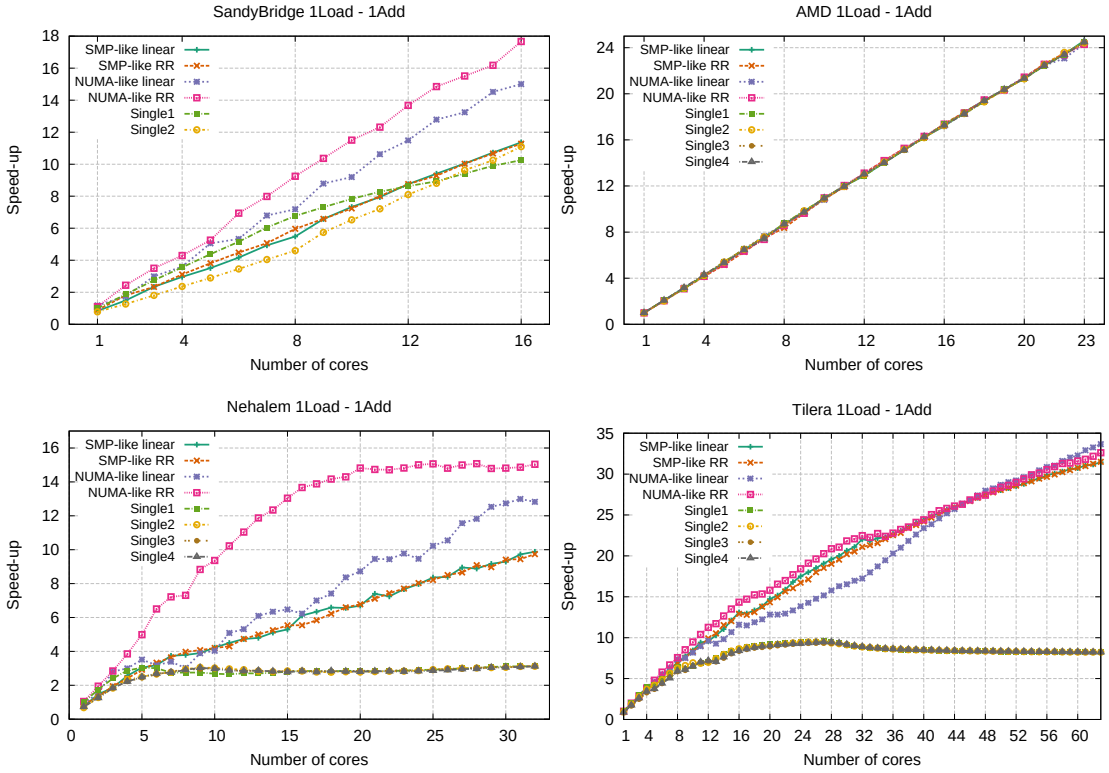


Figure 6.5: Hypothetical Speed-up of a farm parallelization, 1Load-1Add per iter.

times. This is indeed expected, as the thread is allocated on the first chip, near the first memory controller. This difference (reported in numbers in Table 6.3) is related to the extra time spent through the external interconnection network. The average time of **Single2** suddenly drops at 9 threads, because when we start allocating threads on the second chip, these threads have a smaller R_q that beneficially affect value averaged among all the threads. The behavior on **Single1** is mirrored: we can easily see a change in the inclination after 8 cores; this can be mistakenly considered a hint that the memory interface is becoming a bottleneck, but it is more likely the effect of using threads on the other cores, that have therefore an increased R_q w.r.t this interface.

The distance of the memory controller also affect the SMP-like allocation, where data is uniformly distributed between the two interfaces, increasing the average R_q since the beginning.

Unsurprisingly, a NUMA-like allocation in which each thread always access the nearest memory offers better results, ending with a time 36% lower than the SMP policy.

Table 6.3: Synthetic Benchmark results on **SandyBridge**, 1Load-1Add.

	Time per iteration (clock cycles)					
Worker	SMP		Numa		Single	
	Linear	RR	Linear	RR	Mem1	Mem2
1	230.6	218.8	163.6	175.4	192.8	248.4
2	258.6	213.4	220.4	158	206.75	304.8
4	260.6	247.8	214	179.6	217	326
8	281.2	258.6	214.6	166.8	227.8	335.4
9	264	263.4	197.4	167.4	237.2	302.2
15	269.6	271	199.2	178.8	292.2	282.2
16	271.4	273.4	205.6	174.6	300.8	278

AMD Results on this architecture (Table 6.4) seems an anomaly, as basically all the allocation policies, of both memory and processors, give the same performance: all the results fall between 114 and 126 cycles of iteration time. We assist a slight improvement in performance when multiple cores are used, probably because of a parallel memory controller that is able to (partially) serve some requests concurrently. The main result is that in this benchmark the memory is not a bottleneck as its R_q do not increase for any number of cores used.

Nehalem The execution on this architecture is particularly interesting, as we are finally able to observe the behavior in a case where the memory system *is* a bottleneck. With a small number of cores we observe the very same behavior of

Table 6.4: Synthetic Benchmark results on **AMD**, 1Load-1Add.

	Time per iteration (clock cycles)							
Worker	SMP		Numa		Single			
	Linear	RR	Linear	RR	Mem1	Mem2	Mem3	Mem4
1	125.6	124.2	124.2	126.4	125	126.2	126.2	124.4
2	120	122.2	119.8	122.4	120.6	123.4	119.6	120.6
6	116.2	119.4	114.8	118.2	115.6	115.2	114.8	116.2
12	116.4	114.4	115.4	114.4	116.4	115.6	115.4	115
22	117.4	117	119.2	117.6	117.6	116.6	117.4	117.6
23	116.8	117.8	118.2	118.4	117.2	117.8	117.6	117.4

the other Intel architecture, with the NUMA allocation providing the best results when a single core is used. When the number of cores is increased, a single memory interface is not sufficient, and the iteration time grows almost linearly: with 8 cores we have an iteration time of 342.8cycles on *Single1*, that increase of 2.06 times with 16 and of 3.51 times with 32. The behavior of the two NUMA-like policies is highly influenced by the process allocation: in particular, in the linear case, we observe peeks at 8, 16 and 24 after which the iteration time slowly decreases. This is to be expected, as a linear allocation means that with the first 8 threads we completely fill a chip, and produce the maximum memory pressure for the related controller. An analysis of Table 6.5 confirm the fact, showing that the iteration time with 8, 16, 24 and 32 is basically the same, around 320 cycles. After that peek, for example with 9 threads, the average time decreases, because there is one single thread on the second memory controller, that obtain a smaller time w.r.t to the other 8 and decrease the average. As the second chip is filled, of course, the time increases again, reaching the very same value with 16 threads. On the other hand, the NUMA-like RR policy exploit the full aggregate bandwidth of the controllers by allocating processes in a round-robin fashion, resulting in the best times throughout the whole benchmark.

For the SMP-like policies, process allocation is not really important; their execution time is higher a small number of cores but these are able to outperform - at least in some cases, the NUMA-like linear configuration, as they exploit all the memory controllers with any amount of cores. At the end, however, their result with 32 cores are 30% and 50% higher w.r.t, respectively, NUMA linear and NUMA RR.

Tilera The results on the Tile*Pro*64 (Table 6.6) are qualitatively the same of the **Nehalem** machine: a single memory controller is not sufficient and becomes a bottleneck with more than 8 cores. With a single thread NUMA-like policies offer the same time of Single1, while the SMP-like ones result in a higher iteration time. In this case we see more clearly that a NUMA-like linear policy is not really effective when a limited number of cores is used, as even the SMP-like policies perform well. At the end of the graph, with 63 nodes, NUMA-like RR is the best configuration,

Table 6.5: Synthetic Benchmark results on **Nehalem**, 1Load-1Add.

	Time per iteration (clock cycles)							
Worker	SMP		Numa		Single			
	Linear	RR	Linear	RR	Mem1	Mem2	Mem3	Mem4
1	168.8	145.4	133.8	120.8	126.4	185.2	170.8	173
2	172	190	166	130	144.8	197.2	184.8	201.4
4	197.2	210.2	168.8	131.2	176.2	216.2	227.2	227.6
8	267	254.6	335.6	138.4	367.8	342.8	346.6	347.6
9	291.6	279.4	292.6	128.8	413.4	372.2	384.6	375
16	330.8	364.8	324.6	148	708.6	712.8	707	713.8
17	338.8	368.2	307	154.8	761.2	772	759.6	772.2
24	380.8	377.8	320.6	202.2	1045.2	1058.6	1048.8	1070.2
25	378.2	384.8	309	209.8	1082.6	1080	1084.6	1105.6
31	403.2	415	301.6	263.6	1258.8	1260	1264.6	1270.6
32	409.4	415.4	315.4	269	1293	1295.4	1289.4	1297.8

Table 6.6: Synthetic Benchmark results on **Tilera**, 1Load-1Add.

	Time per iteration (clock cycles)							
Worker	SMP		Numa		Single			
	Linear	RR	Linear	RR	Mem1	Mem2	Mem3	Mem4
1	120.8	120.8	109	109	109	116	132	125
4	123.2	122	110.2	114	111.6	118.2	133.2	127
8	125.6	126	118.2	115.2	137	139.4	149	147.8
16	133.2	135	150.6	121.6	202.4	201.8	209	322.2
32	158.2	165.4	202.4	155.2	394.2	393	392.2	392
60	212.8	212.6	202.6	207	796	793	793	791
63	218.4	218	204	210.6	838.4	835.2	835.6	834

but here the SMP-like allocations are not so bad, offering a very slight deterioration w.r.t NUMA RR. At the end, with 63 threads, NUMA allocations performs only a $3-6\%$ better than the SMP ones.

10 Load, 1 Add per iteration

Using a single load we found a bottleneck in two of our architectures. We therefore decided to *further increase* the memory pressure, to be able to observe the behavior even in **SandyBridge** and **AMD**. We decided to execute 10 Loads before issuing the **mfence** instruction. We do not know, precisely, the amount of concurrent memory requests supported by each architecture, so in general we expect that *some* of these ten requests will be serialized. Nevertheless, this will increase as much as possible

the amount of requests received by the memory subsystem.

The results for the iteration time reported in Figure 6.6 confirm our expectation: now three of the four architectures show a bottleneck on the memory. With the AMD architecture the trend remains the same, confirming the singularity of this architecture. We observe that the undulatory behavior of NUMA-like linear is more prominent, confirming the problem of this configuration when partially using the multi-cores. The behavior of SMP-like and NUMA-like RR policies is now more *similar*, with the NUMA allocation still outperforming the others, but usually of a very small factor.

As before, we report in Figure 6.7 the speed-up of a hypothetical parallel program based on this benchmark. Results definitely show that the **NUMA-like RR** is the best configuration, giving the best result throughout the whole graph. We should notice, however, that no matter which configuration is used, the speed-ups obtained are generally very small w.r.t the number of used cores, meaning that a different memory/process allocation policy can increase the performance when the memory is the bottleneck of the system, but only of a limited degree. What we believe a more interesting result is that again **NUMA-like RR** is able to reach the best scalability much sooner w.r.t. the others, meaning that we can save a significant number of cores to achieve the same performance.

SandyBridge A detailed analysis of the benchmark on this architecture, using also numerical values reported in Table 6.7 prompt some important considerations.

The behavior with a single core is still the same, with NUMA-like equivalent to Single1 and considerably outperforming SMP-like. An interesting (but strange) behavior is that in this case SMP-like linear and Single2 shows a clear peek with 8 cores. This is a bit unexpected: in the SMP case, as previously said, linear or RR process allocation policies do not affect the amount of requests per memory controller. Similarly, it is quite difficult to imagine that R_q of the single memory controller (Single2 case) with 8 cores is higher than with 16. Our hypothesis is that in these two cases the bottleneck is not the memory, but the interconnection network between the two chips or the shared level cache on-chip: a linear allocation means that we have all the 8 cores allocated on the first chip, and the memory allocated completely or partially (depending on the case) on the second chip. The iteration time with 8 workers of NUMA linear and Single1 coincides as expected, because the two policies provide the very same process/memory configuration. Using all the cores, the NUMA allocation provides an iteration time 18% smaller than the SMP allocation.

AMD Results on this architecture remains a real mystery: even performing 10 Loads per iteration, the memory is not a bottleneck. This is really strange, and let us think that the benchmark is not working as expected on this architecture. We can, however, definitely say that these 10 requests are partially executed concur-

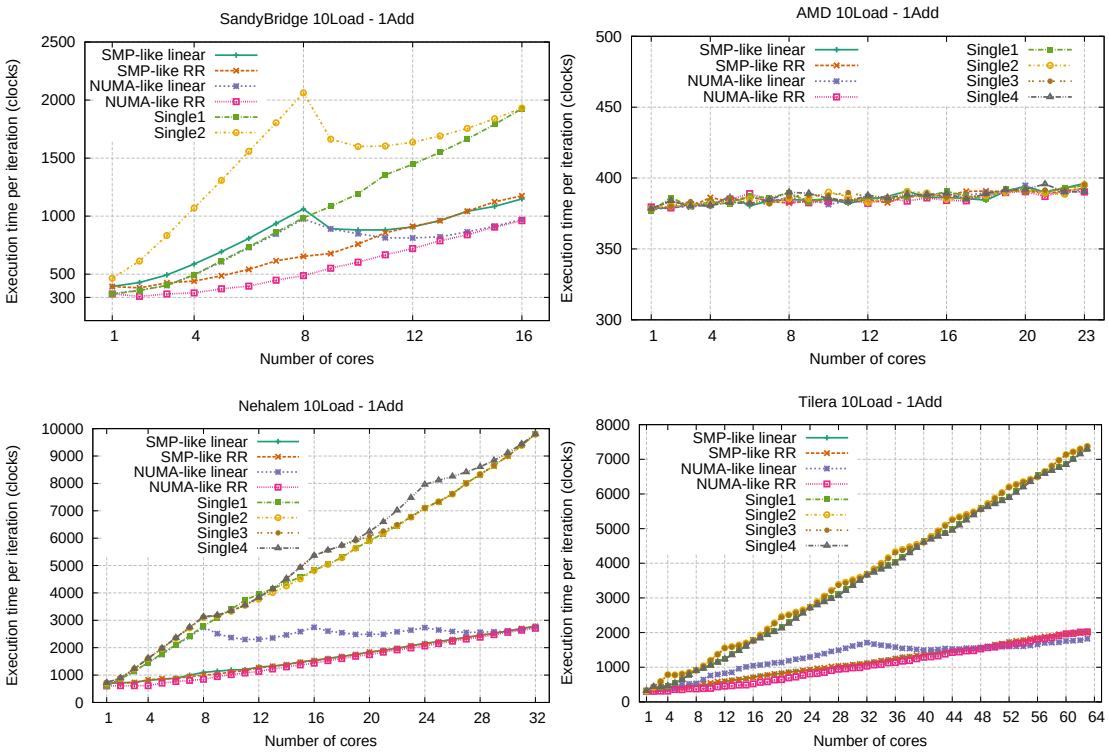


Figure 6.6: Average execution times per iteration, 10Load-1Add per iter.

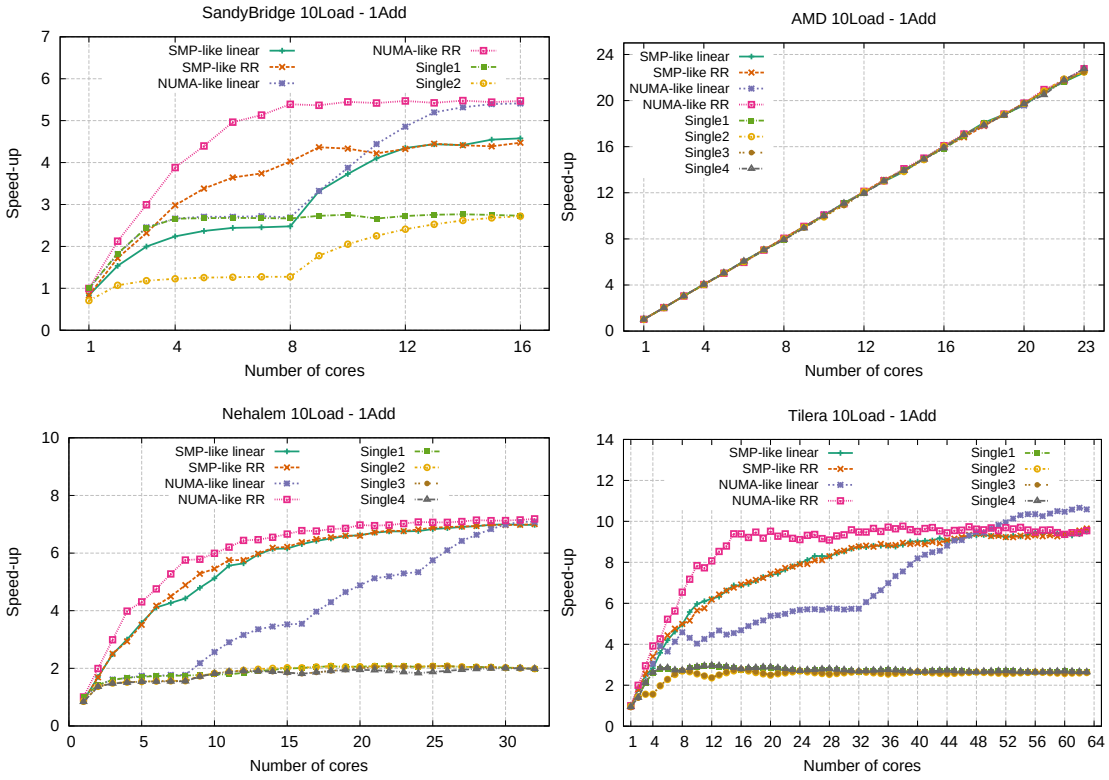


Figure 6.7: Hypothetical Speed-up of a farm parallelization, 10Load-1Add per iter.

Table 6.7: Synthetic Benchmark results on **SandyBridge**, 10Load-1Add.

Worker	Time per iteration (clock cycles)					
	SMP		Numa		Single	
	Linear	RR	Linear	RR	Mem1	Mem2
1	394	393.6	330.4	329.8	328.6	465
2	428	382.2	362	309.2	359	612.8
4	587.2	440.8	493.4	338.6	494	1070.2
8	1060.8	653.2	976.8	487.8	985.8	2061.8
9	891.4	677.6	889.6	551.2	1083.6	1663.6
15	1084	1123.6	913.6	905.6	1791	1839.2
16	1148.4	1175.4	971.4	961.2	1925	1931.6

Table 6.8: Synthetic Benchmark results on **AMD**, 10Load-1Add.

Worker	Time per iteration (clock cycles)							
	SMP		Numa		Single			
	Linear	RR	Linear	RR	Mem1	Mem2	Mem3	Mem4
1	379	377.8	377.6	379.6	376.8	378.6	377.8	378.6
6	380.4	387	386.6	389	387.2	385.2	382.2	382.2
23	396.2	392.8	392.6	390.2	394.6	395.4	395.4	390.4

rently, as the iteration time is only $\sim$ 3 times higher w.r.t having a single Load. This architecture definitely requires a deeper study to better understand its memory subsystem. For the sake of completeness we report some numerical values in Table 6.8.

Nehalem This test confirm the previous results, especially on the behavior of the NUMA-like linear policy, that is definitely unusable in this case. The lighter difference between SMP and NUMA allocations let us introduce another, important consideration: when the amount of requests enqueued in the memory interface is very high, the "distance" between the memory and the cores is still present in R_q, but becomes a really small factor w.r.t the waiting time on the queue. Therefore, if the memory traffic is really high, the usage of local or remote memories become a negligible aspect on the final performance. In this case we are talking of a 2% difference in the final iteration time using 32 cores (Table 6.9). This does not mean, however, that the two policies are interchangeable: by looking at Figure 6.7 we can immediately observe that the NUMA-like RR policy allow us to reach the same speed-up using a smaller number of cores w.r.t the SMP-like polices.

Table 6.9: Synthetic Benchmark results on **Nehalem**, 10Load-1Add.

Worker	Time per iteration (clock cycles)							
	SMP		Numa		Single			
	Linear	RR	Linear	RR	Mem1	Mem2	Mem3	Mem4
1	696.6	700.6	605.6	603.6	608.2	710.6	725.8	723.8
2	715.6	720.8	866	610.2	853	885	892.4	891.6
4	810.8	828.4	1457.8	610.6	1453.8	1601	1599.2	1615.4
8	1099.2	995.8	2748.4	844.8	2778	3101.8	3114	3130.8
9	1141.6	1036.4	2513	946	3101	3153.8	3173.6	3188.6
16	1542.2	1525.8	2743.6	1435.6	4832.8	4815.8	5365.4	5366.4
17	1609.8	1594.2	2605.4	1528.6	5043.8	5040.6	5560	5556.8
24	2157.6	2140	2733.8	2062.2	7105.6	7098	7096.8	7958.4
25	2225.4	2210.2	2645.4	2152	7339.4	7328.2	7330	8113
31	2700.6	2700.4	2660.6	2636.4	9389	9385.4	9389.8	9442.6
32	2788.8	2778.6	2747	2708.4	9797.6	9795.8	9797.2	9801.2

Tilera As the **Nehalem**, we assist in an intensification of the phenomenon already described in the previous results. Again, the time difference between NUMA-like RR and SMP decreases when using the whole set of cores; in this case we obtain basically the same results, making the two solutions equivalent when using 64 cores. As with the other architectures, however, using less cores gives a significant performance improvement on NUMA w.r.t SMP policies.

Finally, an interesting result here is that the NUMA-like linear policy for some reasons is able to outperform all the others by a significant factor ($\sim$ 10% faster than the others). We have no real explanations for that; as with the **AMD** results, this should deserve a deeper study to understand the behavior.

Table 6.10: Synthetic Benchmark results on **Tilera**, 10Load-1Add.

Worker	Time per iteration (clock cycles)							
	SMP		Numa		Single			
	Linear	RR	Linear	RR	Mem1	Mem2	Mem3	Mem4
1	320.2	320.2	306.6	306.4	306.4	313.8	336.8	322.4
4	433.6	361.4	401.8	312.6	472.4	784.4	784.8	467.2
8	493.6	490.8	534.4	374.6	911.6	910	900.6	900.8
16	713.8	708.6	1046	522.2	1778.2	1774.2	1754.4	1753.2
32	1120.8	1117	1709.6	1034.4	3699	3681.8	3652.8	3653
60	1965.8	1962.8	1756.2	1963	6869.4	7135.6	7135	6842.4
63	2032	2000.4	1823.8	2021.4	7311.6	7370.8	7376.2	7296.6

6.2.2 Concluding Remarks

Summarizing all these results and comments, this Synthetic Benchmark allowed us to study the behavior of many multi-core commercially available w.r.t the Memory and Process allocation policies introduced in Section 6.1. The three main points that we understood by this analysis are the following:

- It seems that current architectures are actually well balanced in terms of memory bandwidth per core, so that we need very fine-grained algorithms, or with small locality, to be able to observe a performance deterioration related to the memory subsystem.
- The **NUMA-like** memory allocation policy delivered the best results throughout all the tests, but only if paired with a **Round-Robin** process allocation: the **linear** allocation is not feasible because of the not optimum distribution of requests among the memory interfaces.
- The performance difference between **SMP-like** and **NUMA-like** exists, but in many cases is very small; this difference *may not* justify a program rewrite if we already have a SMP-oriented parallel program; however, when writing a new program (or implementation) the developer *should* prefer the **NUMA-like** memory allocation policy.

6.3 Farm parallelization of the Sobel Operator

We continue the analysis with a more concrete algorithm, that should give us a better idea of the real behavior w.r.t synthetic benchmarks. We need a code to stresses the memory, so it should involve a small amount of calculation for each memory load/store. To this purpose we selected an **Edge Detection Filter**. This kind of algorithm, in short, analyze an image to detect object boundaries inside the picture; this is usually the first step of more complex algorithms to detect, for example, objects inside images and/or movements in videos.

More formally, an edge detector looks for discontinuities in the image brightness, that correspond to changes of depths, of materials, or other conditions that usually represent a border within two different objects inside the image.

To detect these discontinuities, the algorithms works with grey-scale images (as we are only interested in the brightness of the image, not the colors) and estimate the *gradient of the image intensity function*. From this estimation we can easily identify discontinuities in all those points that have a high gradient value.

The **Sobel operator** is one of the most simple and common gradient estimator, able to compute an approximate value for the gradient in each image point in a very simple yet effective way.

Specifically[182], for each point the operator approximate the derivatives $\frac{\delta f}{\delta x}$ and $\frac{\delta f}{\delta y}$ by differences, considering a 3×3 neighboring of the point in the following way:

$$\frac{\delta f}{\delta y} = (x_6 + 2x_7 + x_8) - (x_0 + 2x_1 + x_2)$$
$$\frac{\delta f}{\delta x} = (x_2 + 2x_5 + x_8) - (x_0 + 2x_3 + x_6)$$

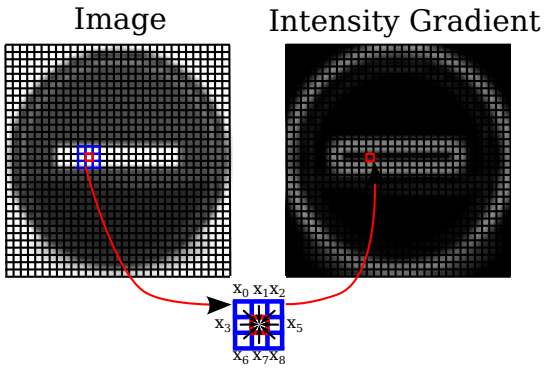


Figure 6.8: Sobel Operator.

From these derivatives approximations we can compute the gradient value in the point as

$$\nabla f = \left|\frac{\delta f}{\delta y}\right| + \left|\frac{\delta f}{\delta x}\right| \tag{6.5}$$

The complete calculation for each pixel therefore consist in 4 multiplications, 15 sums and 2 absolute values. The operation and the required dependencies among pixels is graphically exemplified in Figure 6.8. This is executed once for each pixel of the image, resulting in a linear memory access pattern. The result of the application of the Sobel operator to an image is reported in Figure 6.9.

(a) Original Image (b) Preprocessed Image (c) Gradient Estimation

Figure 6.9: Example of the gradient estimation obtained by the Sobel operator.

6.3.1 Experimental results on the target architectures

To remove other possible sources of degradation we decided to not implement a real farm module, but develop a simplified, farm-like parallel program. The resulting parallel code is similar to the previously discussed Synthetic Benchmark, with a variable number of threads (workers), each one executing, independently, the *Sobel kernel* on a local set of images. Threads synchronize at the beginning with a barrier, to start the computation at the same time. The behavior is the same of a steady-state farm, in which all the workers are computing the Sobel kernel on a different element of the input stream. However, removing input streams and the need of Emitter-Worker and Worker-Collector communications we simplified the code, so that we were able to isolate the behavior of multiple threads executing the Sobel code concurrently.

We ran the farm-like parallelization with images of 512×512 elements, applying the filter to 100 images on each worker. Figure 6.10 show the average filtering time for a single image, varying the number of workers. Corresponding numerical results are in Table 6.11. The behavior of the parallel program as a whole is instead shown with Figure 6.11, where we draw the Speed-up, calculated on the overall throughput (i.e. images per time unit) of the program, compared with the sequential version.

Overall, it seems that this kernel is not able to stress the memory system in three of the four architectures: despite its simplicity, the locality is actually quite good, as each element of the image is brought in cache only once, so that we are not able to produce a high enough number of concurrent requests. This lead to ideal speed-ups on **SandyBridge** and **AMD**, and a practically ideal speed-up of 30 on the **Nehalem** (because of the slight step that occurs on at the beginning of the graph). There are not many considerations for these three architectures, except for the fact that, again, **NUMA-like RR** perform slightly better than the others, and that in some cases we are still able to notice the step when a different memory controller is used (especially on **SandyBridge** with *Single and SMP linear*.

The time increase on the *Nehalem* graph is quite interesting because it seems to happen when more than two workers are allocated on a single chip: all the linear allocations (including the *Single* configurations of course) show a step at 3, while with both the Round-Robin ones it happen with more than 8 workers. It should not be a memory-interface related problem (because it happen regardless of the memory allocation) so it is probably related to the shared level of cache (perhaps its size, or the maximum concurrent requests supported). Anyway, the performance drop is quite small, and do not require further studies.

What, indeed, deserve a better analysis is the **Tilera** behavior, as it seems that, in this case, a single controller is not sufficient and, more interesting, we notice an increasing difference between SMP and NUMA allocations. This is the opposite of what we saw on the Synthetic Benchmark, where the difference becomes smaller as we increase the number of cores.

We executed some more tests with different image sizes. This should not affect

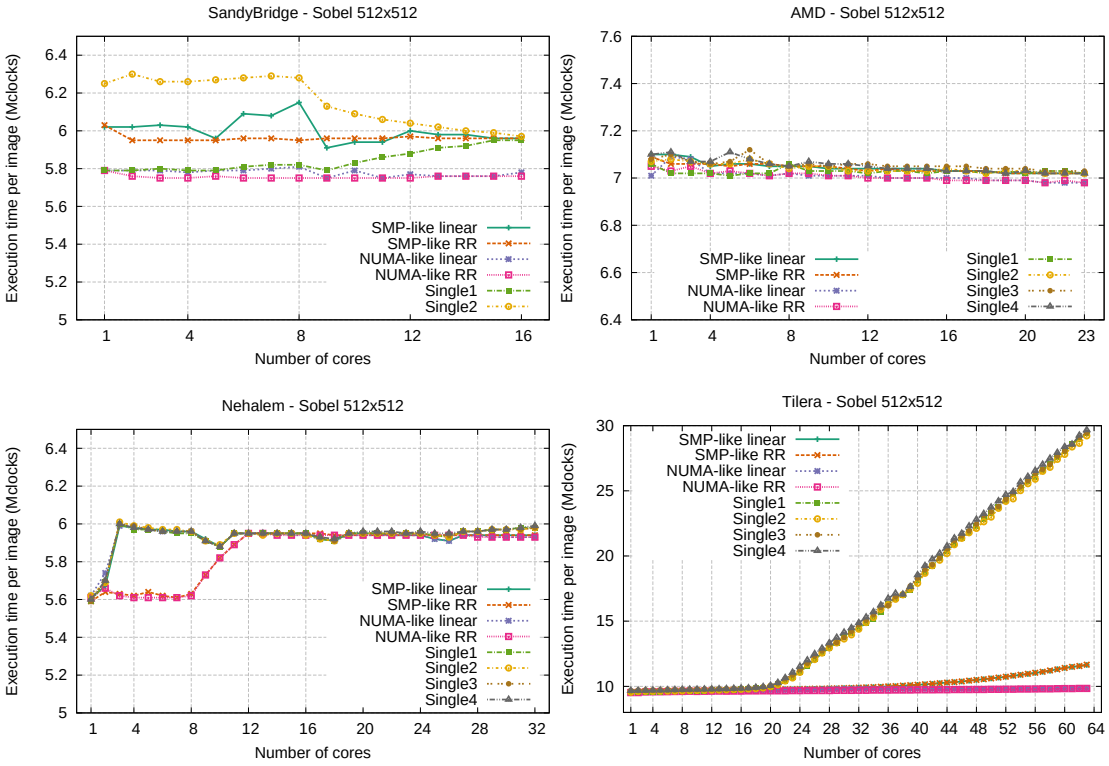


Figure 6.10: Average filtering time per image, 512×512 pixels.

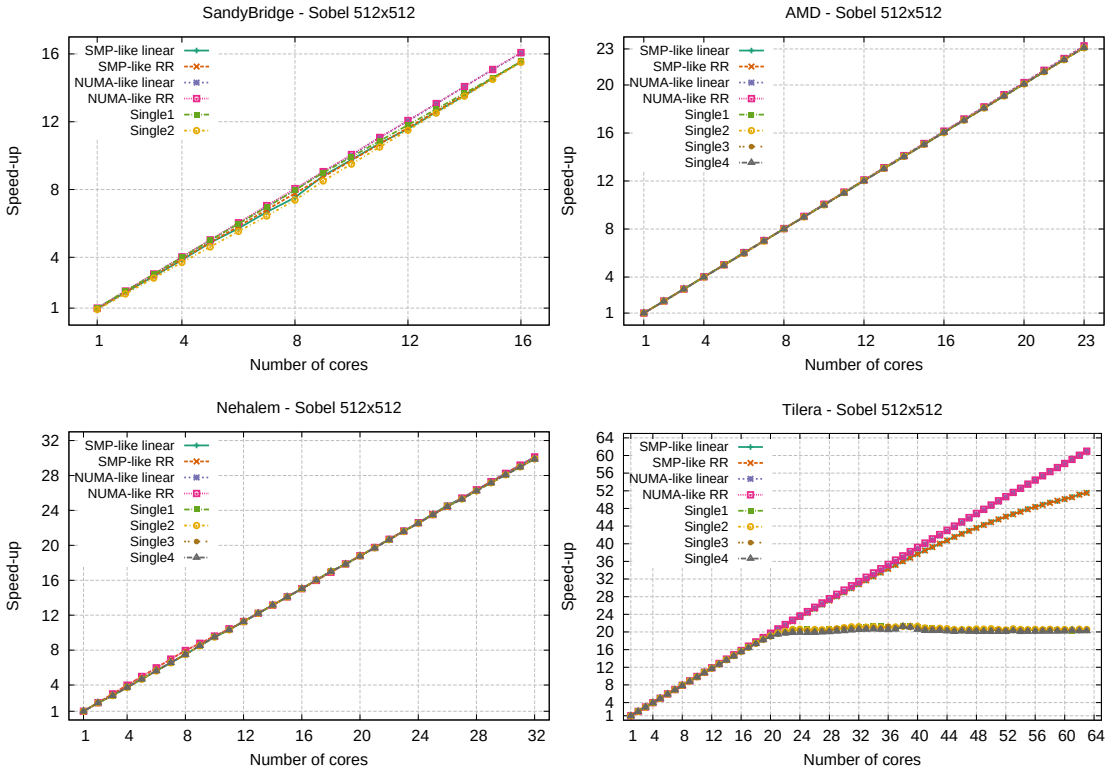


Figure 6.11: Speed-up of the farm parallelization, 512×512 pixels per image.

Table 6.11: Sobel image filtering times for 512×512 images. Times in clock cycles.

SandyBridge

Worker	SMP		Numa		Single	
	Linear	RR	Linear	RR	Mem1	Mem2
1	6.02M	6.03M	5.79M	5.79M	5.79M	6.25M
2	6.02M	5.95M	5.79M	5.76M	5.79M	6.30M
8	6.15M	5.95M	5.81M	5.75M	5.82M	6.28M
16	5.96M	5.96M	5.78M	5.76M	5.95M	5.97M

AMD

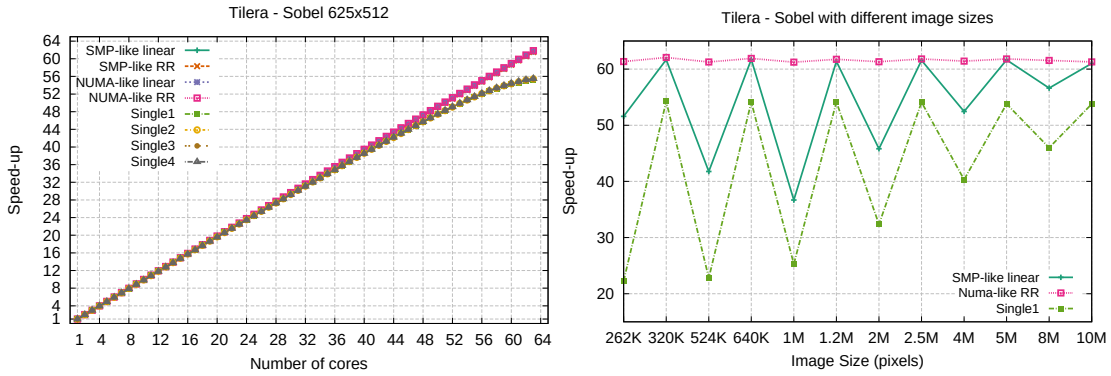
Worker	SMP		Numa		Single			
	Linear	RR	Linear	RR	Mem1	Mem2	Mem3	Mem4
1	7.10M	7.09M	7.01M	7.05M	7.06M	7.07M	7.08M	7.10M
2	7.10M	7.06M	7.09M	7.03M	7.02M	7.08M	7.10M	7.11M
6	7.06M	7.06M	7.02M	7.02M	7.02M	7.08M	7.12M	7.08M
23	7.02M	7.02M	6.98M	6.98M	7.02M	7.02M	7.03M	7.02M

Nehalem

Worker	SMP		Numa		Single			
	Linear	RR	Linear	RR	Mem1	Mem2	Mem3	Mem4
1	5.61M	5.59M	5.62M	5.61M	5.59M	5.62M	5.59M	5.60M
2	5.67M	5.64M	5.74M	5.66M	5.69M	5.69M	5.70M	5.70M
8	5.96M	5.63M	5.96M	5.62M	5.95M	5.96M	5.96M	5.96M
16	5.95M	5.95M	5.94M	5.94M	5.95M	5.94M	5.95M	5.95M
17	5.92M	5.95M	5.92M	5.94M	5.92M	5.92M	5.93M	5.93M
18	5.91M	5.94M	5.91M	5.94M	5.91M	5.91M	5.91M	5.92M
32	5.94M	5.94M	5.93M	5.93M	5.98M	5.98M	5.98M	5.99M

Tilera

Worker	SMP		Numa		Single			
	Linear	RR	Linear	RR	Mem1	Mem2	Mem3	Mem4
1	9.57M	9.57M	9.54M	9.52M	9.53M	9.55M	9.68M	9.67M
2	9.58M	9.57M	9.56M	9.52M	9.56M	9.60M	9.71M	9.68M
4	9.64M	9.64M	9.56M	9.58M	9.59M	9.62M	9.72M	9.69M
8	9.67M	9.68M	9.61M	9.61M	9.65M	9.65M	9.75M	9.75M
16	9.72M	9.72M	9.67M	9.64M	9.78M	9.77M	9.84M	9.85M
32	9.90M	9.90M	9.81M	9.70M	14.48M	14.39M	14.74M	14.87M
48	10.49M	10.49M	9.80M	9.76M	22.30M	22.12M	22.45M	22.80M
62	11.55M	11.56M	9.84M	9.84M	28.97M	28.65M	29.11M	29.25M
63	11.64M	11.66M	9.84M	9.84M	29.42M	29.22M	29.46M	29.67M



(a) Speedup for 625×512 images (b) Speedup with different image sizes, 63 cores

Figure 6.12: Speedup with different image sizes on the Tile*Pro*64.

the amount of memory requests, because we execute the very same operation on each pixel and a pixel is read and wrote only once, so increasing the number of pixels equally increase the number of requests and the amount of calculation per image (i.e. the time between two memory requests remain the same).

By increasing the number of pixels of a small quantity (using images of 625×512 pixels) the behavior, as shown in Figure 6.12a, returns aligned to the other architectures, with a speed-up of ~ 62 with 63 cores. This for both SMP-like and NUMA-like allocations, while using a single controller we experience a small slowdown starting with 40 cores. We tried different sizes and the general behavior using 63 cores is reported in Figure 6.12b: images composed of a perfect square number of pixels becomes a problem. This is probably related to the specific memory access pattern, that may interfere with the caching system and cause a degradation on the memory response time.

6.3.2 Concluding Remarks

In summary, we can safely say that with this algorithm the ratio of memory requests and calculation time is not sufficiently high to pose a problem in current and future architectures.

This can be considered a notable result, given the fact that this is indeed a fine-grained algorithm that only applies some additions and multiplications for each element. Moreover, w.r.t other algorithms working with matrices, the amount of reuse is very low, so the cache subsystem can only exploit locality. Nevertheless, we have no significant performance problems even with the 32 cores of **Nehalem** and the 64 of **Tilera**. Theoretically the **Numa-like** allocation policy is a bit better than the **SMP-like**, but in practice the difference is very small.

The only significant problem emerged on the Tile*Pro*64 architecture, and only with specific configurations (high parallelism degrees and specific image sizes); unfortunately we were not able to certainly determine the cause of the performance drop,

but can consider the result a singularity, probably related to the specific memory access pattern.

6.4 Farm parallelization of the Vector Addition

Given the results with the *Sobel operator*, we selected a further finer grained problem. We selected what is probably the finest grain problem one can find in mathematical operations: the addition of two vectors. The pseudo code reported in Listing 6.4 shows the simplicity of the code: for each iteration we execute two loads, one add and one store. This is also an important benchmark because of its simple loop structure, that allows modern compiler to produce vectorized code if the architecture support it. Vectorized code is important in our study, as it is able to increase the instruction throughput of a processor and thus, in our case, the number of memory requests per time unit. Of the target architectures, all the x86-based processors include a vector unit able to execute 4 to 8 (depending on the architecture) integer operations per clock; the Tile*Pro*64 architecture, instead, do not support natively vector instructions; however its VLIW instruction set allow the execution of two arithmetic operation per clock, resembling, in some way, a short-vector operation.

```
for (i=0;i<n;i++)
        y[i] = y[i] + x[i];
```

Listing 6.4: Vector Addition Pseudo code.

6.4.1 Experimental results on the target architectures

Experiments were conducted using the same pseudo-farm parallel program introduced with the Sobel experimentation. We execute the addition over two vectors of 1024000 elements, and each worker execute the addition 100 times. As before, we report the average calculation time per vector in Figure 6.13 and Table 6.12, and the parallel speed-up measured w.r.t the sequential version that uses the nearest memory controller in Figure 6.14.

We consider the results very positive as this application qualitatively mimics the synthetic benchmark previously studied. We find most of the characteristics already studied in the previous benchmark, in particular:

- The wavelike behavior of the **NUMA-like linear** policy, because of the limited memory bandwidth exploited when we partially use the architecture
- The generally higher latency in **SMP-like** allocations, that makes the program slower with small parallelism degrees even w.r.t using a single memory interface

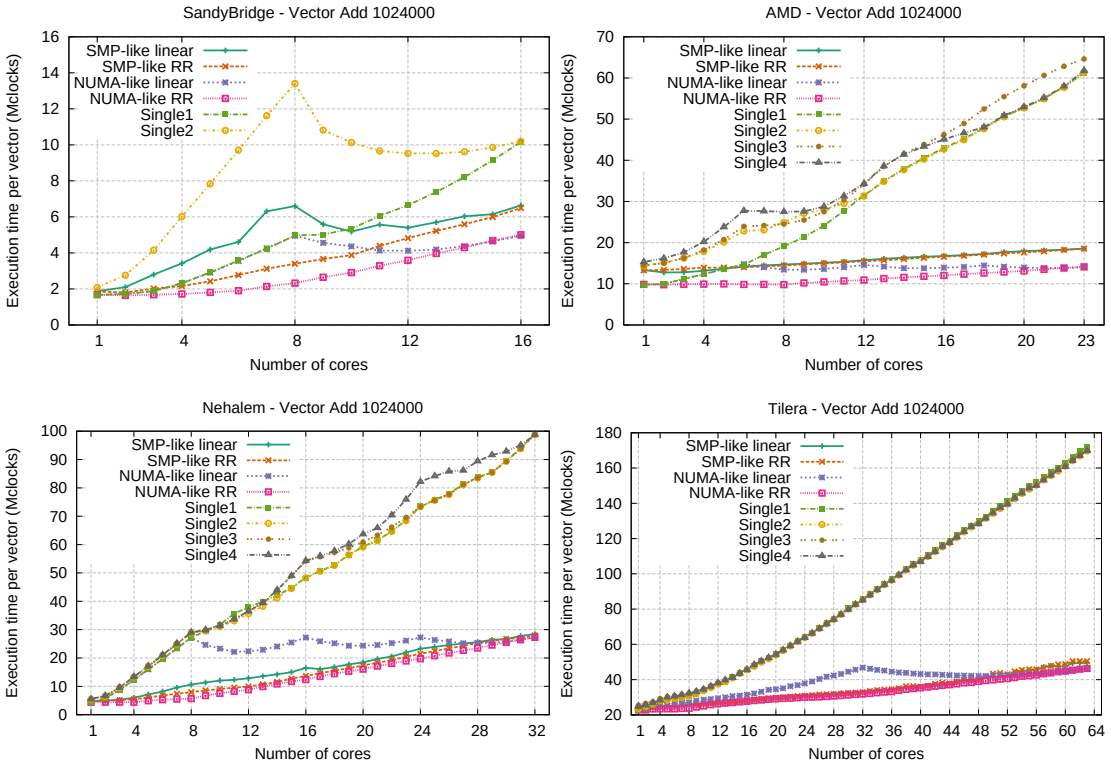


Figure 6.13: Average execution times per addition, 1024000 elements per vector.

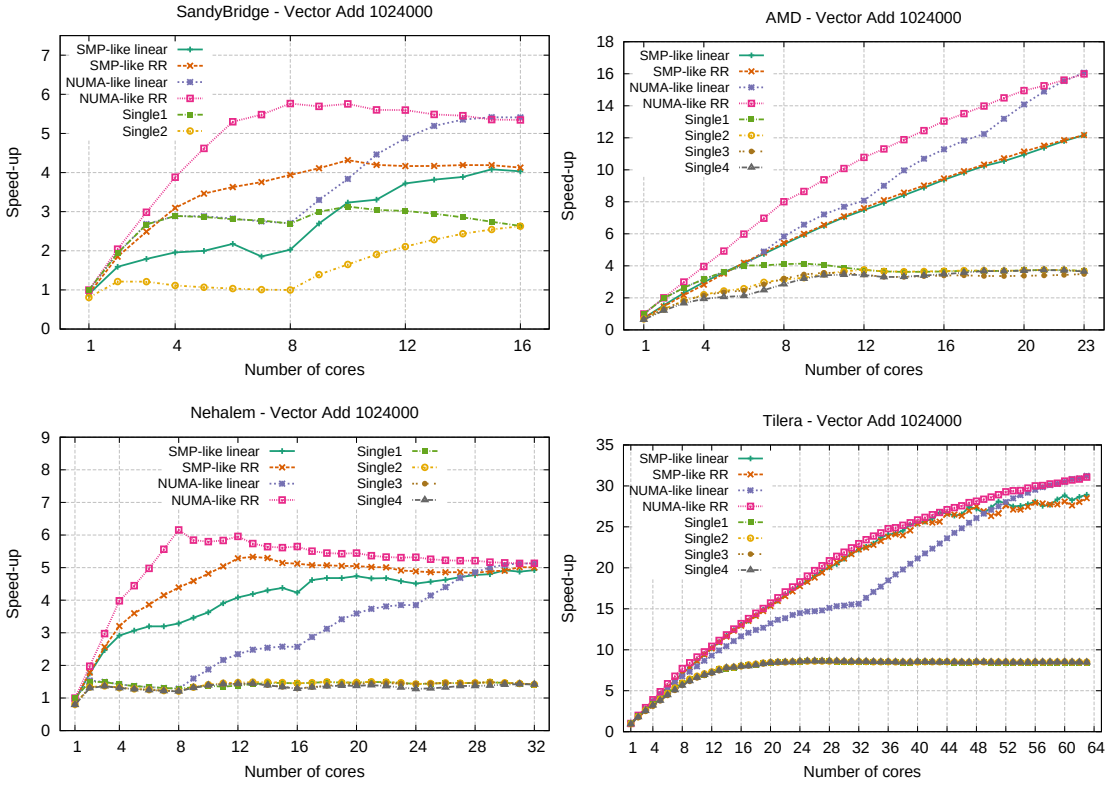


Figure 6.14: Speed-up of the farm parallelization, 1024000 elements per vector.

Table 6.12: Vector addition times for 1024000 elements vector. Times in clock cycles.

SandyBridge

Worker	SMP		Numa		Single	
	Linear	RR	Linear	RR	Mem1	Mem2
1	1.87M	1.85M	1.68M	1.68M	1.67M	2.08M
2	2.10M	1.81M	1.72M	1.64M	1.72M	2.76M
8	6.60M	3.40M	4.95M	2.32M	4.97M	13.40M
16	6.64M	6.49M	4.95M	5.01M	10.16M	10.19M

AMD

Worker	SMP		Numa		Single			
	Linear	RR	Linear	RR	Mem1	Mem2	Mem3	Mem4
1	13.33M	13.28M	9.79M	9.85M	9.81M	14.35M	14.44M	15.30M
2	12.76M	13.37M	9.97M	9.77M	9.94M	15.18M	15.01M	16.17M
6	14.15M	14.04M	14.78M	9.84M	14.64M	22.83M	23.95M	27.75M
12	15.73M	15.47M	14.57M	10.92M	31.45M	31.33M	34.13M	34.32M
23	18.55M	18.54M	14.04M	14.11M	61.16M	61.39M	64.61M	61.78M

Nehalem

Worker	SMP		Numa		Single			
	Linear	RR	Linear	RR	Mem1	Mem2	Mem3	Mem4
1	4.68M	4.72M	4.39M	4.38M	4.37M	5.43M	5.47M	5.53M
2	4.99M	4.91M	5.70M	4.43M	5.84M	6.55M	6.58M	6.70M
4	6.00M	5.45M	12.30M	4.39M	12.33M	13.29M	13.32M	13.39M
8	10.64M	7.97M	27.18M	5.68M	27.16M	28.87M	28.89M	29.04M
16	16.53M	13.66M	27.22M	12.39M	48.35M	48.25M	54.18M	54.33M
24	23.28M	21.48M	27.26M	19.73M	73.47M	73.36M	73.36M	82.19M
32	28.44M	27.96M	27.29M	27.27M	98.96M	98.75M	98.75M	98.75M

Tilera

Worker	SMP		Numa		Single			
	Linear	RR	Linear	RR	Mem1	Mem2	Mem3	Mem4
1	24.08M	24.10M	22.87M	22.84M	22.85M	23.53M	25.51M	24.67M
2	24.20M	24.16M	23.04M	22.84M	24.13M	24.91M	26.56M	25.47M
4	24.72M	24.61M	24.60M	23.43M	27.05M	28.18M	29.31M	28.34M
8	25.78M	25.87M	27.06M	23.71M	30.90M	30.79M	32.34M	32.36M
16	27.96M	28.27M	31.37M	27.70M	45.71M	45.54M	45.98M	45.70M
32	32.93M	32.88M	46.84M	31.86M	85.81M	85.35M	85.21M	85.08M
48	40.27M	39.88M	42.06M	38.97M	129.83M	128.70M	128.85M	128.34M
63	49.75M	50.50M	46.09M	46.32M	171.94M	169.58M	170.16M	169.85M

- The performance superiority of the **NUMA-like RR** policy, that always offer the best performance, although sometimes the improvement w.r.t **SMP-like** policies is limited.

On top of these considerations, we were finally able to verify the existence of a memory bottleneck on the **AMD** architecture.

6.5 Data-Parallel parallelization of the FFT

We conclude the study by introducing a complex Data-Parallel application, representing the other mayor class of parallel programs. Data-Parallel programs differ from the previously seen stream-parallel programs because workers of the parallel application work in synergy to compute a *single* result of the application. This may lead to data sharing and to more complex data exchange patterns between the different workers of the parallel program. For this study we selected a classic algorithm: the **Fast Fourier Transform**.

A fast Fourier transform (FFT) is an algorithm to compute the discrete Fourier transform and its inverse, thus converting time (or space) to frequency and vice versa. As a result, fast Fourier transforms are widely used for many applications in engineering, science, and mathematics. **FFT** is also a quite interesting algorithm in the area of parallel computing because of its data-parallel version that involves a variable stencil: data dependencies among workers changes on every step of the algorithm.

The algorithm works with real numbers, and because of this most implementations are designed to exploit the floating point units of the processors. Nevertheless, because of its wide use, the algorithm has been deeply studies also for embedded and DSP processors that, sometimes, are not equipped with floating point units. As a result fixed-point implementations with a limited error rate exists in literature; this is quite important in our case because of the absence, on the Tile*Pro*64 architecture, of a floating point unit.

The pseudo code of the original sequential FFT algorithm is shown in Listing 6.5, while in Figure 6.15 we show the dependency pattern among iterations for an 8-point transform. The code uses the points array x plus an auxiliary array w that contains unit radices; the algorithm works in-place, reading and writing completely the array x in each of the $\log(N)$ iterations (N is the size of the input array). This algorithm is very interesting because the data array x is not read with a linear pattern: x_{n2} and x_{n3} are not contiguous points, as depicted in Figure 6.15, and their distance changes at each iteration (creating the variable stencil). Yet the code, with slight changes, can be vectorized with a proper optimizing compiler to take advantage of the vector units of x86 processors. This, coupled with the lightweight calculation per point, is likely to produce a code that, as the vector sum, stresses the memory, making the use of multiple controllers important for the overall performance.

```
void fft(x, w, N){
  for(q=0;q<log2(N);q++){
    L = 1; L <<= (q+1);
    r = 1; r <<= (t-(q+1));
    L2 = L>>1; kL = 0;
    for (k=0; k<r; k++) {
      for (j=0; j<L2; j++) {
        n3 = kL + j;
        n2 = n3 + L2;
        n1 = L2 - 1 + j;
        temp = w[n1]*x[n2];
        x[n2]=(x[n3]-temp)/2;
        x[n3]=(x[n3]+temp)/2;
      }
      kL += L;
} } }
```

Listing 6.5: FFT Pseudo code.

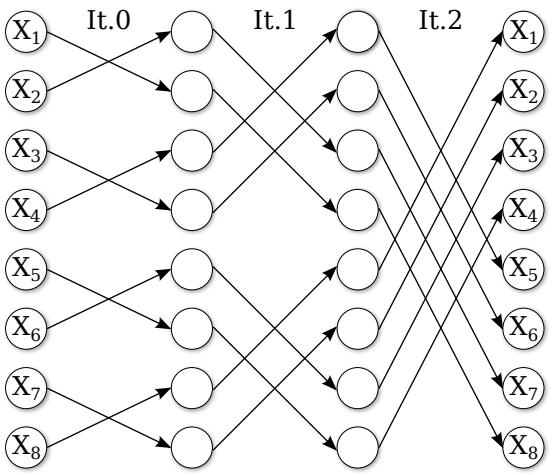


Figure 6.15: Data dependencies in FFT.

6.5.1 Parallel FFT

For our study we implemented the parallel FFT by using the well-known *Binary Exchange Algorithm*. In fact, two main families of parallelizations are known for the FFT[90]. We selected the *Binary Exchange* because it is a direct parallelization of the sequential algorithm in Listing 6.5, while the other family (the *Transpose Algorithm*) is derived from a different sequential code. Furthermore, the *Binary Exchange Algorithm* is a notable example of a data-parallel with variable stencil, making it quite interesting to cover an example of this class of algorithms. However, more complex parallelizations exists in literature. A good review of algorithms for performing FFT on multicore, and a performance comparison, can be found in [81], where the authors present Spiral, an automatic code generator for computing Fourier transforms.

A simple yet effective parallel version can be achieved by a simple (static) partitioning of the input array x. At the maximum theoretical parallelism degree each partition is composed of a single point. We have that, for each iteration, we require a communication to receive a point, and another communication to send our point. In particular, given point x_h, at iteration i we will require the value of point x_k with $k = h + 2^i$ if $h < k$, $k = h - 2^i$ otherwise. This makes the stencil quite interesting because when point x_h requires the value of point x_k, we can simply verify that point x_k requires the value of x_h; this means that in the maximum theoretical parallelism the communication is symmetric: we need to receive a point from the worker that is computing x_k, and send our value to the very same worker. When using larger partitions, with multiple points per worker, if the two corresponding point (i.e. x_k and x_h) are owned by the same worker, no communications are required at all for these two points. Conceptually, we could try to find an algorithm that minimizes

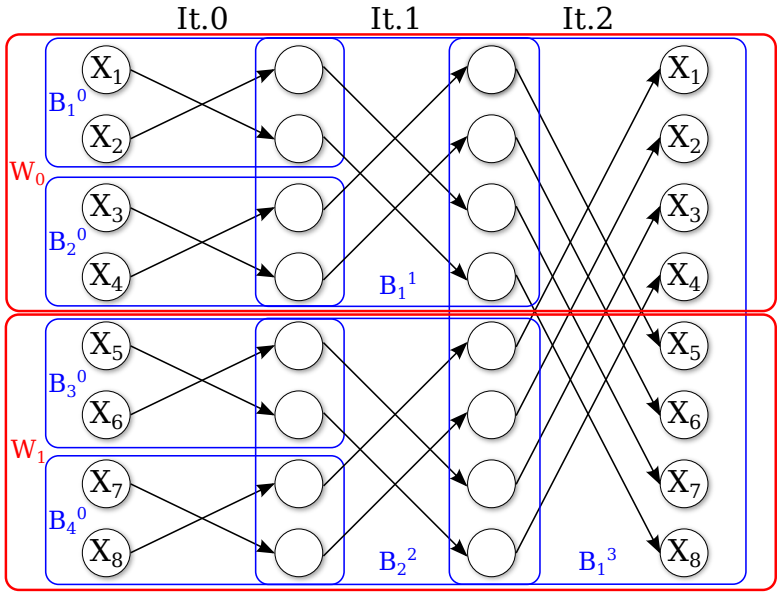


Figure 6.16: Data dependencies and blocks in a statically partitioned parallel FFT.

the communications, by keeping as much as possible x_k and x_h in the same partitioning. Because of the variable stencil, however, we know that, at some point, the two elements will reside in different partitions. A good static partitioning consist in creating partitions of contiguous elements (i.e. *slices* x), such that, for the first iterations, all the couples $\langle x_k, x_h \rangle$ are contained inside the same partition. After a certain number of iterations, however, *each element* of the partition will require a value from a different partition, as in Figure 6.16.

Keeping x_k and x_h in the same partition is also beneficial for the computation, as the *temporary* values of the sequential algorithm (Listing 6.5) can be computed once per couple; otherwise, if x_k and x_h reside in different workers, each worker will have to compute the temporary value. This means that in a parallel version, beside the communication and/or synchronization time required to exchange data between workers, we also have an increased computation time, that will affect the performance, and thus the scalability: we cannot really expect an ideal scalability for this parallel algorithm, as the amount of calculation is increased as the number of workers grows.

Another important aspect of the sequential code is its vectorization possibilities, that makes the inner loop of the computation vectorizable by current compilers. For this reason we wanted to keep (as much as possible) the same sequential structure inside each worker. For the sake of simplicity, we call the inner loop a *Block Calculation*. Now, each partition may, of course, contains a variable number of blocks, depending on the amount of elements per partition and the current iteration (blocks are selected respecting data dependencies and thus its number and size changes at each iteration, as shown in Figure 6.16).

A block can be composed of points contained in a single partition (and in this case we can exploit both the reuse of temporary values *and* vectorization) or divided between multiple partitions. In this second case we cannot reuse the temporary values, but we can still exploit, to a limited degree, the vectorization.

In general, each worker will calculate, at each iteration, a number of *blocks*; the first and the last one *may* be partial (i.e. divided with other partitions and thus workers), while the other, inner blocks are, by definition, complete. The pseudo-code of a worker will therefore adhere to the structure of Listing 6.6: for each iteration we will compute the first partial block (if existing); then a number of complete blocks and, at the end, the last, partial, block. After this, a data exchange procedure is taken to receive and distribute data for the following iteration.

In our case we decided to write a shared-memory implementation of the parallel algorithm. At each iteration we require the data from the previous one, causing problems in our in-place algorithm because the data may be overwritten by the owner worker before being read by the others. We therefore relax the original in-place semantics of the algorithm, by defining a *source array* and a *destination array*. The source array will contain the values from the previous iteration, while the destination one is updated with the values computed in the current iteration. At the end of an iteration, the destination array becomes the source array of the following iteration; the current source, instead, contains data that will never be used again, and becomes the next destination array. The resulting pseudo-code of the shared-memory implementation is described in Listing 6.7. A hypothetical message-passing version, instead, will require explicit send and receive to exchange data. In this case the resulting code (reported in Listing 6.8) is a bit more complicate as it requires sending and receiving the data of the partial blocks to and from the other workers. Then, if we want to overlap communications with computation, we can begin to work on the complete blocks (if available); finally, we will wait the missing data of the partial blocks to compute them.

If we use dynamic partitioning techniques (such as re-partitioning the array at each iteration, or partition the array dynamically throughout the whole computation) these problems, of course, can be removed and/or mitigated, at the cost, however of moving the data during the computation. In this chapter, however, we are just interested in comparing the same parallelization with different allocation policies: the fact that the parallelization chosen may not be the best achievable is not really a concern here.

6.5.2 Experimental results on the target architectures

We implemented a portable version the algorithm in Listing 6.7, so that we were able to run the program on the four target architectures. However, the FFT code needed some tuning to efficiently run on the various processors. In particular, the most important problem of FFT is the use of real numbers, commonly implemented with a floating point representation. Unfortunately, the Tile*Pro*64 does not support

```
Worker i:
  for (q=0; q<log2(N); q++) {
    Compute First Partial Block //if exist
    Compute Complete Blocks
    Compute Last Partial Block //if exist
    Data Exchange/Synchronization
  }
```

Listing 6.6: Parallel FFT. Generic Worker pseudo code.

```
Worker i:
  for (q=0; q<log2(N); q++) {

    Compute First Partial Block
    Compute Complete Blocks
    Compute Last Partial Block
    Barrier
    Src-Dst Buffer Exchange

  }
```

Listing 6.7: Shared Memory FFT Worker.

```
Worker i:
  for (q=0; q<log2(N); q++) {
    Send First Partial Block
    Send Last Partial Block
    Compute Complete Blocks
    Receive First Partial Block
    Compute First Partial Block
    Receive Last Partial Block
    Compute Last Partial Block
  }
```

Listing 6.8: Message Passing FFT Worker.

natively floating point operations, that must be emulated in software. In our case this is a major problem, because we need a fast code if we want to be able to stress the memory and compare the different policies. Fortunately, the problem of computing the FFT is very common in digital signal processing (DSP). In most DSP appliances computations are performed by mean of embedded processors, that (at least in the past) rarely implement a floating point unit. For this reason several studies exists on FFT algorithms over *fixed point* real numbers. The algorithm, and the resulting approximation error, has been studied and optimized for fixed point operations, so that currently exists fast FFT algorithms working on 16-bit fixed point real numbers[137, 161, 179]. Using such version let us keep the parallel structure uniform and obtain a fast FFT on any architecture.

An important difference in this application is that, because of the data-sharing among workers, we cannot (easily) use non-coherent memory areas for the computation; we therefore enabled automatic cache coherence even on the **Tilera** architecture.

Differently from the previous code, in this case the amount of calculation per worker decrease when the parallelism degree grows, so we need a different metric than the completion time, to better examine the effects of memory contention. We selected the average *calculation time per element*.

Each worker has a static partition assigned, at most of size

$$P_{size} = \left\lceil \frac{N}{n} \right\rceil \tag{6.6}$$

where N is the size of the array x and n the number of workers. Of course if N is not divisible by n, we need to round values, and some workers will have one element more than others. Given the partition size, at each iteration of the algorithm the amount of element processed per worker is exactly P_{size}. Given number of iterations, defined as $k = log_2 N$, we have that each worker, throughout its computation, will process

$$P_{size} * k \simeq \frac{N * log_2 N}{n} \tag{6.7}$$

elements. The sequential completion time is, of course, defined as

$$T_c^{seq} = T_{elem} * N * log_2 N \tag{6.8}$$

So if we assume a balanced computation, in which each worker takes the same time per iteration, we can estimate the theoretic completion time of the parallelization as

$$T_c^n = \frac{T_{elem} * N * log_2 N}{n} \tag{6.9}$$

meaning that, as expected, in absence of performance degradation, T_{elem} should remain constant varying the parallelism degree. While the memory contention is one of the most important sources of performance degradation, here we also expect a slight increase in the average T_{elem} because of the increased computation given by having partial blocks. In general, however, a graphical study of the variations in T_{elem} should allow us to easily notice the degradation induced by the memory response time. For the sake of clarity, we calculate T_{elem} starting from the completion time as follows

$$T_{elem}^n = \frac{T_c^n * n}{N * log_2 N} \tag{6.10}$$

Of course, to give an idea of the overall parallelization result we will also present the scalability

$$S^n = \frac{T_c^{seq}}{T_c^n} \tag{6.11}$$

Given the previous results, for the sake of readability we will present only one configuration with a single memory controller, plus the four possible allocation policies.

FFT on 1048576 points

We start analyzing the performance results of the FFT on an array of $N = 2^{20} = 1048576$ complex elements. Average computation time per element is shown in Figure 6.17, while the scalability is in Figure 6.18 and some numerical results are reported in Table 6.13.

As intuitively expected, the execution time per element tend to increase when using multiple workers. However we can easily notice two different behaviors: in

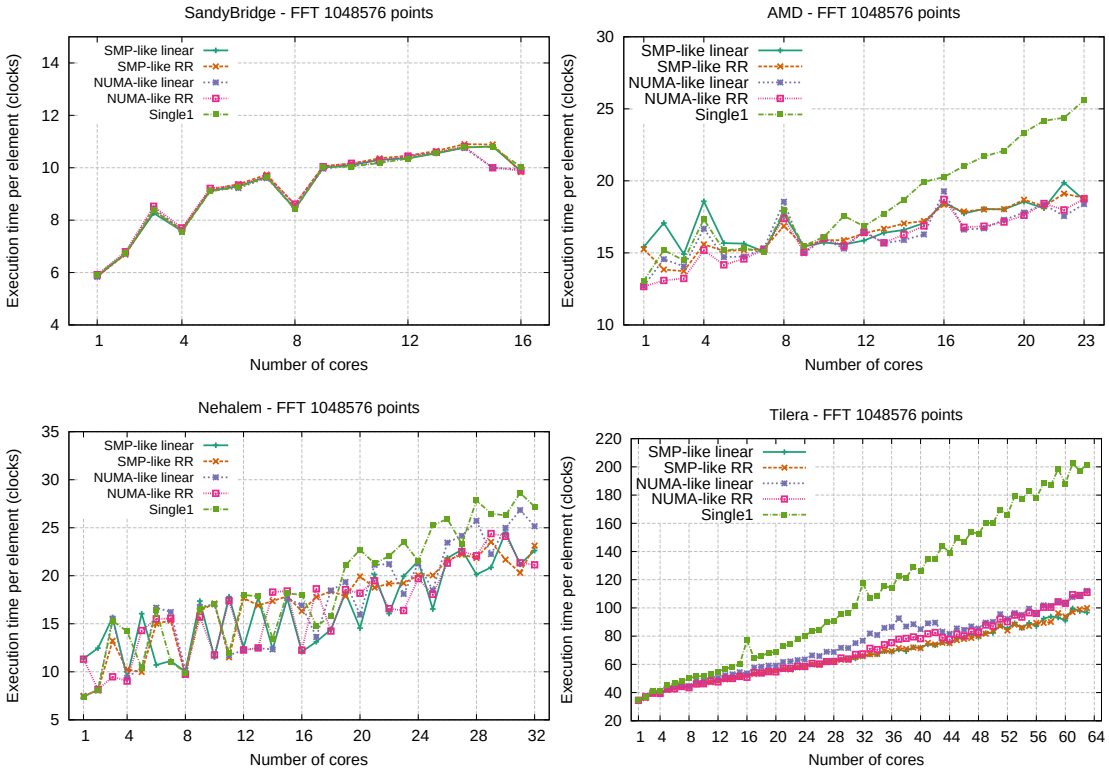


Figure 6.17: Average calculation time per element - FFT on 1048576 points.

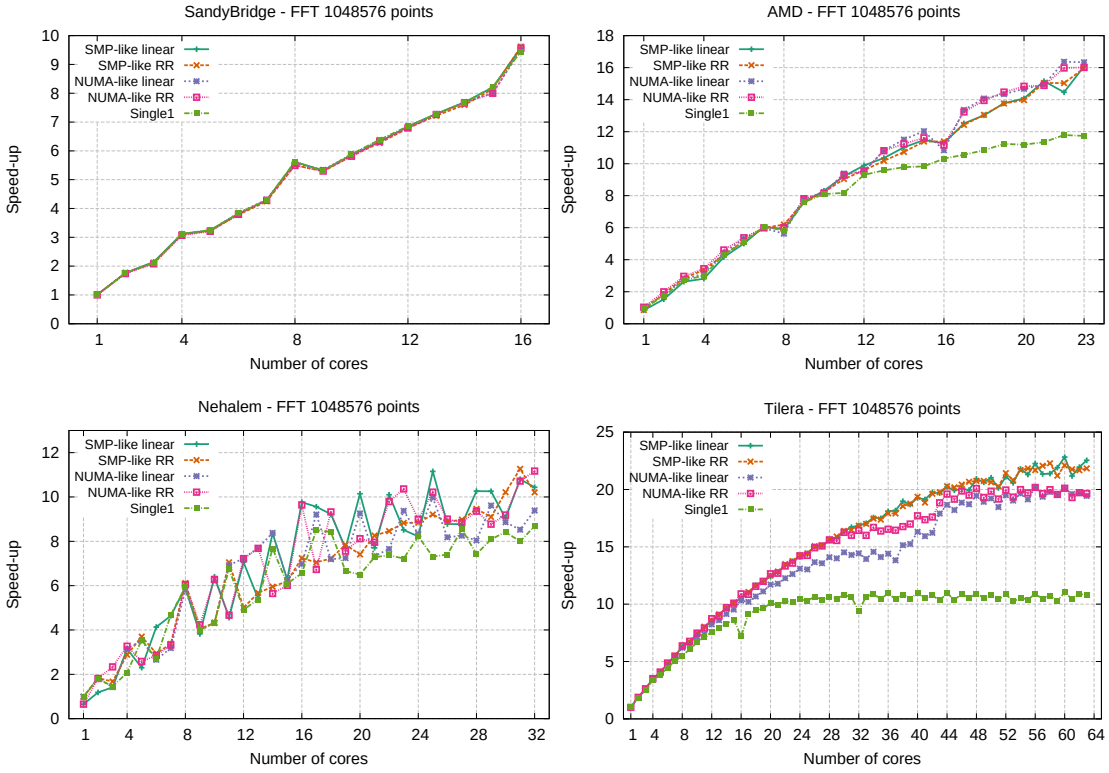


Figure 6.18: Speed-up of the data-parallel FFT - 1048576 points.

Table 6.13: Calculation time per element - FFT on 1048576 points. Times in cycles.

SandyBridge

Worker	SMP		Numa		Single
	Linear	RR	Linear	RR	Mem1
1	5,85	5,87	5,84	5,92	5,91
2	6,73	6,69	6,72	6,80	6,74
3	8,27	8,36	8,46	8,53	8,38
4	7,59	7,65	7,61	7,70	7,56
5	9,12	9,17	9,13	9,21	9,10
8	8,43	8,60	8,49	8,61	8,41
16	9,87	9,84	9,95	9,89	10,03

AMD

Worker	SMP		Numa		Single
	Linear	RR	Linear	RR	Mem1
1	15,42	15,26	12,71	12,67	13,06
2	17,07	13,85	14,56	13,09	15,20
3	14,92	13,73	14,07	13,23	14,48
4	18,58	15,59	16,68	15,19	17,34
5	15,68	15,13	14,71	14,18	15,17
16	18,54	18,35	19,28	18,71	20,26
23	18,68	18,82	18,39	18,76	25,59

Nehalem

Worker	SMP		Numa		Single
	Linear	RR	Linear	RR	Mem1
1	11,38	7,50	7,44	11,31	7,38
2	12,46	8,06	8,13	8,16	8,15
3	15,66	13,23	15,54	9,48	15,30
4	9,39	10,23	9,37	9,04	14,22
8	10,01	9,66	10,16	9,75	9,93
16	12,10	16,31	16,92	12,25	18,01
32	22,62	23,13	25,15	21,15	27,19

Tilera

Worker	SMP		Numa		Single
	Linear	RR	Linear	RR	Mem1
1	34,43	34,29	34,62	34,55	34,61
8	44,15	43,90	44,69	43,49	50,26
16	52,70	52,43	53,66	50,81	77,01
32	65,67	65,75	76,75	67,40	117,45
33	66,98	67,18	81,93	71,40	107,27
63	96,73	99,88	112,22	110,99	201,56

the case of **SandyBridge** and **Nehalem** all the five configurations - including the single memory controller - show the same increase, while on **AMD** and **Tilera** the time increase using a single memory is much higher w.r.t using multiple interfaces. The second is the clear sign of a performance degradation related to a memory bottleneck. In general, as always, the **NUMA-like RR** perform equal or better than the other, except in the **Tilera** case when, for the first time, an **SMP-like** allocation offer a slightly better scalability.

The reason of this marked performance difference between architectures is probably due to size of the data set of the application. With $N = 2^{20}$ we have a memory occupation (considering the space needed for two copies of x plus a copy of w) of $12MiB$, so for the x86 architectures the whole data set should be able to stay in the cache hierarchy without the need of re-reading data from the memory in the various iterations. On the contrary, for the Tile*Pro*64 this value is larger than the total amount of L2 caches of the architecture. This explain the difference between Intel-based processors and the Tile*Pro*64, but the AMD architecture show an unexpected behavior. It could be related to the behavior of the L3 cache, that works as a *victim cache*, and to the implementation of the inter-chip cache coherency mechanism.

It is also interesting noticing the sudden changes when the number of workers is a power of two. This is indeed related to the algorithm, as in this cases the number of incomplete blocks is reduced. Yet the behavior is strange because, while in the **SandyBridge** architecture this result in a performance increase, in the others (especially **AMD** and **Tilera**) the results show a drop in the performance. The different results are probably related to the efficacy of the cache system, and in particular the set-associative mechanism, that may produce more conflicts when addresses are aligned to powers of two.

FFT on 16777216 points

Given the absence of a bottleneck in some architectures, we decided to increase the number of points of x. We selected a number of points $N = 2^{24} = 16777216$, resulting in a data set of $192MiB$. In this case none of the proposed architectures is able to keep the whole data set in cache, so we expect this to be a better test-bed for our experiments. For the sake of conciseness, given the previous results of the **Tilera** machine, we decided to omit this architecture from the test as we already analyzed the under-load behavior. Instead, we present a comparison between vectorized and non-vectorized code on the **SandyBridge**, to show the effectiveness of vector instructions in presence of a memory-intensive application.

Average computation time per element is shown in Figure 6.19, while the scalability is in Figure 6.20 and some numerical results are reported in Table 6.14.

It important to verify that, as expected, the time per element in the sequential version is very similar w.r.t the previous configuration of 2^{20} points, because the memory is not a bottleneck and the amount of calculation per element is not influenced by the number of points.

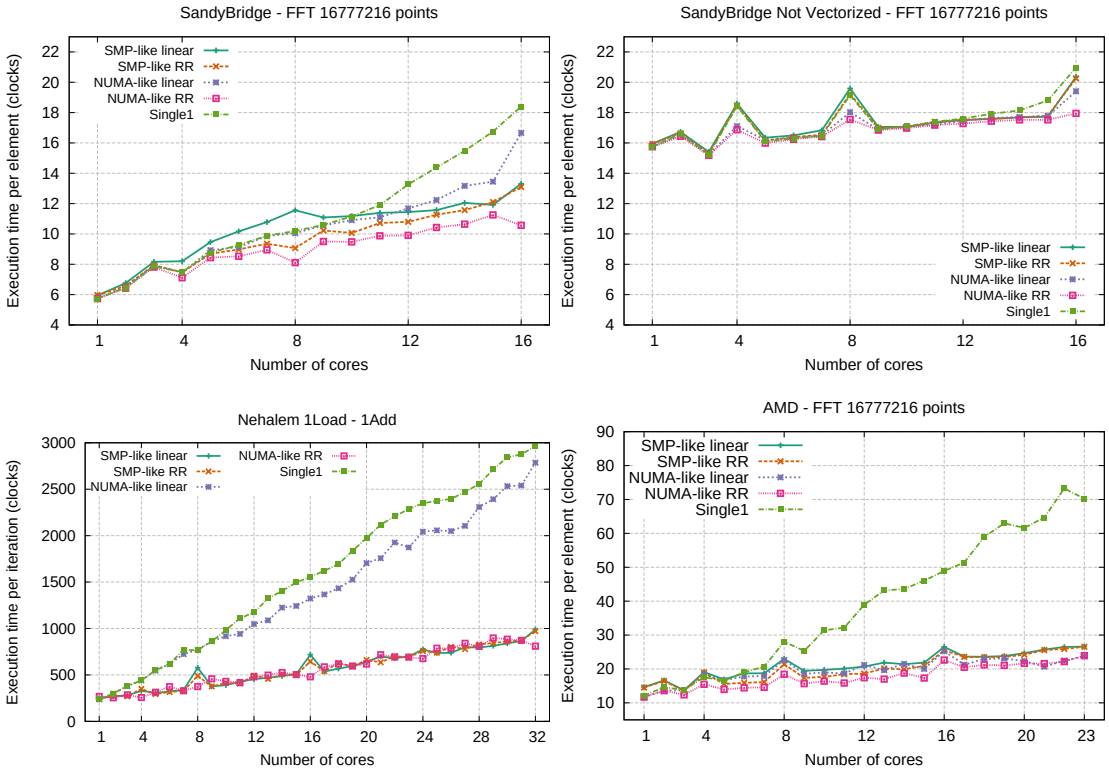


Figure 6.19: Average calculation time per element - FFT on 16777216 points.

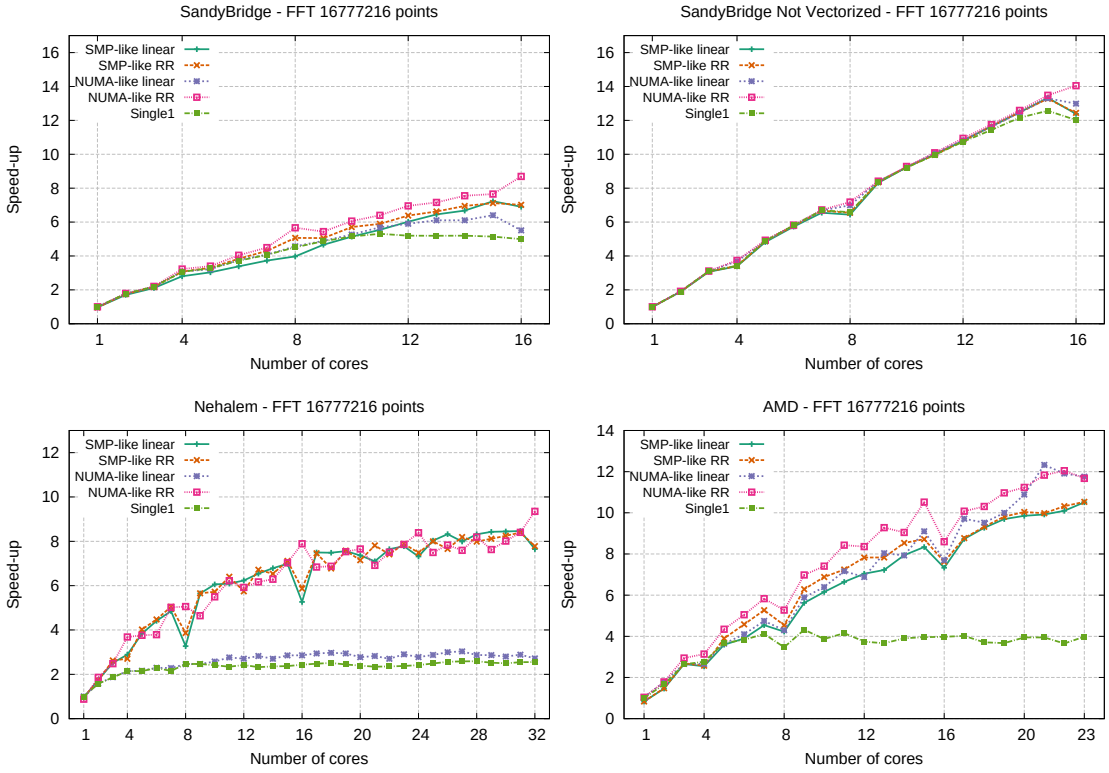


Figure 6.20: Speed-up of the data-parallel FFT - 16777216 points.

Table 6.14: Calculation time per element - FFT on 16777216 points. Times in cycles.

SandyBridge

Worker	SMP		Numa		Single
	Linear	RR	Linear	RR	Mem1
1	5,95	5,96	5,75	5,74	5,74
2	6,76	6,58	6,50	6,42	6,47
4	8,20	7,46	7,46	7,11	7,49
5	9,46	8,69	8,93	8,44	8,77
8	11,56	9,07	10,05	8,11	10,19
16	13,31	13,10	16,67	10,57	18,40

SandyBridge - Not Vectorized

Worker	SMP		Numa		Single
	Linear	RR	Linear	RR	Mem1
1	15,95	15,93	15,75	15,75	15,75
2	16,73	16,64	16,53	16,43	16,60
4	18,59	18,48	17,11	16,87	18,42
5	16,35	16,16	16,19	15,99	16,10
8	19,59	19,24	18,03	17,55	19,14
16	20,36	20,26	19,41	17,95	20,97

AMD

Worker	SMP		Numa		Single
	Linear	RR	Linear	RR	Mem1
1	14,65	14,54	11,69	11,73	12,16
2	16,62	16,46	13,69	13,60	14,60
3	13,75	13,56	13,70	12,38	13,81
4	19,09	18,91	18,27	15,49	17,58
5	16,93	15,59	16,68	13,99	16,54
16	26,53	25,51	25,26	22,63	48,93
23	26,59	26,54	23,78	23,94	70,23

Nehalem

Worker	SMP		Numa		Single
	Linear	RR	Linear	RR	Mem1
1	10,41	10,15	9,83	11,19	9,85
2	11,27	11,42	12,45	10,57	12,50
4	13,65	14,58	18,37	10,70	18,37
5	12,85	12,25	22,78	13,10	22,93
8	24,09	20,40	32,07	15,58	31,99
16	29,94	26,88	55,16	19,97	64,85
32	41,21	40,50	116,07	33,71	123,43

First of all, using vector instruction set we obtain a $\sim 3\times$ improvement on the sequential performance; this improvement, however, tend to decrease when using multiple cores, as the memory becomes the bottleneck. An interesting result is obtained by using a single memory controller, where the difference tend to disappear using all the cores: the vectorized code takes ~ 18 clock cycles per element, while the non-vectorized one only three cycles more (~ 21). This also affects the scalability: we can easily see that the non-vectorized code has an almost-linear scalability (14 with 16 cores), while the vectorized code has a much lower value (9 at most).

In general, now all the architectures highlight the problem of memory bandwidth. As in the previous benchmarks, **NUMA-like, linear** proves to be an ineffective policy, resulting in this case (especially for the Intel architectures) in a performance very similar to using a single memory controller throughout the whole graph. **NUMA-like, RR** works very well, obtaining, in general, the best results.

6.5.3 Concluding Remarks

To summarize, the differences of this data-parallel benchmark w.r.t the previous stream-parallel ones affect the memory subsystem in many ways: data sharing and the use of cache coherence mechanisms on the Tile*Pro*64 are some examples. Yet, the performance results of the various allocation policies are confirmed, showing that the **NUMA-like, RR** policy is generally the best when the memory is a bottleneck. However, the **SMP-like** policies are, again, very competitive, often offering very similar results. In particular, for the first time, we experienced a case in which in fact an SMP-like policy offer better results than the NUMA-like RR, confirming again that the choice is not so straightforward in many cases.

6.6 Modeling policies in the architectural model

In Part II we studied cost models in order to predict the performance of a parallel program running on a target architecture. In particular, with Chapter 5 we defined a Queueing Network model for the Tile*Pro*64 architecture. The model, however, was defined and verified by using a single memory interface of the Tile*Pro*64. Multiple interfaces are handled by specifying routing probabilities for every processor, according to the program semantic. While this can be difficult for a generic program, if the program uses only the **memory allocation policies** identified in this chapter, we are able to straightforwardly select the correct probabilities:

- **Single**: in this case each processor has probability $p = 1$ of sending the request to the specific memory interface of the policy, and a probability of 0 for the other interfaces.
- **NUMA-like**: each processor has probability $p = 1$ of sending the request to the nearest memory interface, and 0 for the other interfaces.

- **SMP-like**: each processor has probability $p = 0.25$ of sending the request to any interface, as the data is uniformly distributed among all the memory controllers. While this is obviously correct for real SMP architectures; here, given the page-base interleaving, this can be considered an approximation. In fact, *the overall probability*, throughout the execution, is 0.25, but subsequent requests of a processor will usually belong to the same virtual page, and thus reach the same memory. Yet if we consider the system as a whole, it is very likely that, in a specific moment of the execution, each memory interface will be used by one fourth of the processors, thus verifying the assumption.

The **process allocation policy**, on the other hand, is easily obtained by selecting the correct processor on the QN model, so that the L_{Net} respect the real processor-memory distance.

We verify the probabilities by taking one of the experimental results on the Tile*Pro*64 and comparing them w.r.t results obtained by simulating the QN model using EQNSim. For the sake of simplicity we selected the first synthetic benchmark, with 1Load and 1Add per iteration. In this case we can easily estimate the service time of the processor queues of our model to $T_p = 20\tau$; given the presence of the **mfence** instruction, the measured iteration time is given by $T_{iter} = T_p + R_q$, so we can easily compare the measured R_q with the result of the QN model.

We start with a simple yet effective analysis on the iteration time with a single worker. In this case we do not need the QN as the memory queue is surely empty. The first worker is allocated in core 0 that, according to the network latency model has the following mesh latencies (with L^x_{net} we indicate the latency for memory x):

$$\begin{aligned} L^1_{net} &= 2(1+1-1)+21+3-4 = 22\tau \\ L^2_{net} &= 2(4+1-1)+21+3-4 = 28\tau \\ L^3_{net} &= 2(4+8-1)+21+3-4 = 42\tau \\ L^4_{net} &= 2(1+8-1)+21+3-4 = 36\tau \end{aligned} \tag{6.12}$$

The resulting T_{iter} will therefore be composed as follows:

$$\begin{aligned} T^{single_1}_{iter} &= T_p + T_s + L_{mc} + L^1_{net} \\ T^{single_2}_{iter} &= T_p + T_s + L_{mc} + L^2_{net} = T^{single_1}_{iter} + 6\tau \\ T^{single_3}_{iter} &= T_p + T_s + L_{mc} + L^3_{net} = T^{single_1}_{iter} + 20\tau \\ T^{single_4}_{iter} &= T_p + T_s + L_{mc} + L^4_{net} = T^{single_1}_{iter} + 14\tau \\ T^{numa}_{iter} &= T_p + T_s + L_{mc} + L^1_{net} = T^{single_1}_{iter} \\ T^{smp}_{iter} &= T_p + T_s + L_{mc} + \frac{L^1_{net} + L^2_{net} + L^3_{net} + L^3_{net}}{4} = T^{single_1}_{iter} + 10\tau \end{aligned} \tag{6.13}$$

In Table 6.15 we report again the T_{iter} results obtained by the experimentation, where we can find the same differences that we estimated when using a single worker (with an error of at most 3 cycles).

Table 6.15: Synthetic Benchmark results on **Tilera**, 1Load-1Add.

	Time per iteration (clock cycles)							
Worker	SMP		Numa		Single			
	Linear	RR	Linear	RR	Mem1	Mem2	Mem3	Mem4
1	120.8	120.8	109	109	109	116	132	125
4	123.2	122	110.2	114	111.6	118.2	133.2	127
8	125.6	126	118.2	115.2	137	139.4	149	147.8
16	133.2	135	150.6	121.6	202.4	201.8	209	322.2
32	158.2	165.4	202.4	155.2	394.2	393	392.2	392
60	212.8	212.6	202.6	207	796	793	793	791
63	218.4	218	204	210.6	838.4	835.2	835.6	834

For the under-load iteration time we used the load-only load dependent queue studied in the previous chapter. The numerical results for the most important configurations are reported in Table 6.16, while on Figures 6.22 and 6.21 we can appreciate a graphical comparison. The estimation is quite good, offering an average error of $\sim$ 6% when using a single controller, $\sim$ 2.2% when using the SMP-like allocation and $\sim$ 5.2% when using the NUMA-like one. This confirm the general applicability of the model, that was obtained with data from a completely different program on the Tile*Pro*64 simulator.

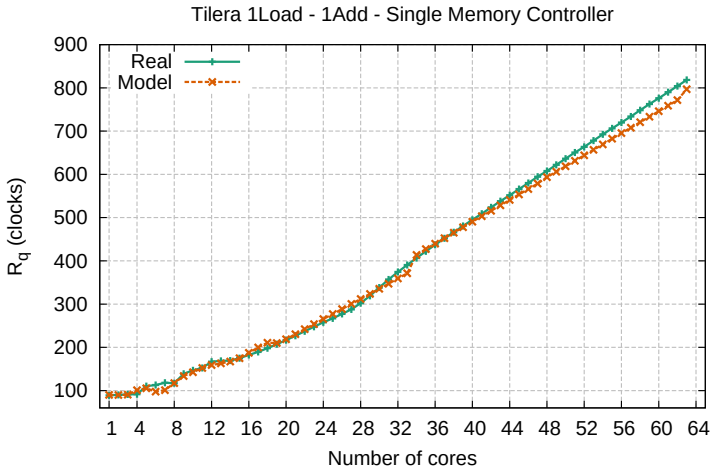


Figure 6.21: Graphical comparison of real and estimated Memory R_q, single int.

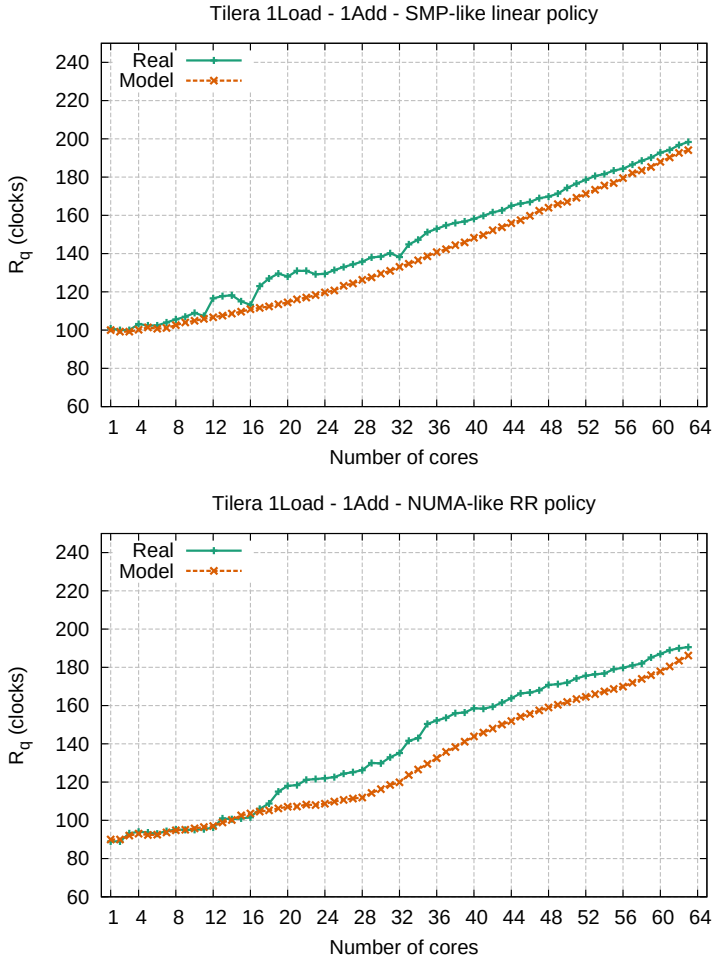


Figure 6.22: Graphical comparison of real and estimated Memory R_q, multiple int.

Table 6.16: Comparison of real and estimated Memory R_q. Times in cycles.

Worker	Single			SMP-like Linear			NUMA,RR		
	Real	Model	% Err.	Real	Model	% Err.	Real	Model	% Err.
1	89	90,0	1,1	100,8	100,0	0,8	89	90,0	1,1
2	90,2	89,7	0,5	100	99,2	0,8	89	90,0	1,1
3	91	90,6	0,5	100	99,2	0,8	93,2	92,0	1,3
4	91,6	101,0	10,2	103,2	100,2	2,9	94	93,0	1,1
8	117	117,4	0,3	105,6	102,6	2,8	95,2	94,8	0,4
10	146,8	142,4	3,0	109	104,9	3,7	95	95,8	0,8
12	166,8	159,0	4,7	116,6	106,7	8,5	96,2	97,0	0,8
16	182,4	187,4	2,7	113,2	110,9	2,0	101,6	103,6	1,9
18	198	210,6	6,4	127	112,4	11,5	108,8	105,2	3,3
20	216,6	218,7	1,0	128	114,5	10,5	118	107,1	9,2
22	237,2	242,0	2,0	131	117,0	10,7	121,2	108,2	10,7
24	257,8	265,4	2,9	129,4	119,7	7,5	122	108,7	10,9
26	277,2	288,6	4,1	133	123,2	7,4	124,4	110,7	11,0
28	302,2	311,9	3,2	135,8	126,3	7,0	126,2	111,9	11,3
30	338,2	335,4	0,8	138,4	129,5	6,5	129,8	116,3	10,4
32	374,2	359,2	4,0	138,2	133,1	3,7	135,2	119,9	11,3
34	406,4	413,8	1,8	147,2	136,5	7,3	143	126,6	11,5
36	437	439,8	0,6	153	140,8	8,0	152,2	132,5	13,0
38	466	465,0	0,2	156	144,4	7,5	156	138,3	11,4
40	495	490,5	0,9	158,2	148,3	6,3	158,6	143,9	9,2
42	523,6	515,7	1,5	161,6	152,2	5,8	159,4	148,0	7,1
44	551,8	540,9	2,0	165	155,9	5,5	163,8	151,9	7,2
46	580	566,1	2,4	167	159,8	4,3	166,8	155,7	6,7
48	607,8	593,4	2,4	169,8	164,0	3,4	170,8	159,0	6,9
50	636,2	618,7	2,7	174,4	167,1	4,2	172	161,9	5,9
52	663,6	644,0	2,9	178,6	171,3	4,1	175,6	164,6	6,3
54	692,4	669,3	3,3	181,6	175,6	3,3	176,8	167,4	5,3
56	719,8	695,3	3,4	184,4	179,5	2,7	179,8	169,9	5,5
58	748,2	720,6	3,7	188,6	183,5	2,7	182	174,0	4,4
60	776	746,0	3,9	192,8	188,0	2,5	187	177,9	4,9
61	790	758,6	4,0	194,2	190,3	2,0	189	180,5	4,5
62	804	771,6	4,0	196,8	192,7	2,1	190	183,5	3,4
63	818,4	796,8	2,6	198,4	194,1	2,2	190,6	186,2	2,3

6.7 Summary

In this chapter we studied the problem of exploiting the multiple memory controllers available in current multi-core architectures. We introduced different process and memory allocation policies and compared the resulting performance of a set of benchmarks composed of synthetic and real kernels, in stream- and data-parallel programs. Finally, we provided a way of modeling these policies using the architecture cost model of the previous chapters.

The results shows a correlation between memory and process allocation: a memory policy can offer very different results depending on the process allocation chosen. This is particularly important especially when using the **NUMA-like** memory allocation, that requires a specific process allocation to allow the parallel program to exploit the multiple memory controllers independently of the parallelism degree.

Memory allocation offer significant performance differences when using a small number of cores. In this case **NUMA-like RR** is able to offer better response times because of the distance between processors and memories, that is kept smaller w.r.t the **SMP-like**.

As the number of cores grow, the difference between allocation policies tends to decrease, as the processor-memory distance become a small part of the total R_q, dominated by the waiting time. In these cases, **NUMA-like RR** is usually still better than the other policies, but of a limited amount.

The fact that we were not able to find an all-purpose policy is not really a problem here: the important point is that, by using the architecture model, we are able to find the best for the specific characteristics of the parallel program we are producing.

An important side note of this work, that was not studied in the chapter, is the *cost of allocation*: to obtain a fair comparison, in all the cases we allocated the data before the beginning of the computation. In general, the cost of data organization should not be much different in the cases, as it usually consist in copying the data in a new memory area that uses the selected memory policy. Yet, in some cases, some policies could require more copies than the others. For example, in a stream-parallel program, data is received from some external entity. At the reception it may be not clear which process would end using the data, so the **NUMA-like** policies could require another copy when the processor is selected. This may be prevented, of course, in **SMP-like** policies because, regardless of the destination worker, data is always distributed among all the memory interfaces. This additional cost may be important for the overall parallel program, so that the parallel pattern model should take it into account.

Chapter 7

Software-based Cache Coherence

The cornerstone of shared-memory systems is the possibility of cooperating by using the shared memory. However, cache hierarchies introduces private, uncontrolled memory areas between the processor and the memory, leading to the problem of **cache incoherence**, in which multiple caches hold different copies of the same memory location. The mainstream solution consist in preventing incoherence through the use of hardware mechanisms that ensures that each cache hold the *current* value of a memory location. These mechanisms (commonly called cache coherence protocols) make the handling of cache coherence completely invisible to the software level.

Hardware cache coherence has come to dominate the market for technical, as well as for legacy, reasons. When the problem of cache coherence was introduced, several studies tried to introduce and compare hardware-based and software-based cache coherency protocols.

Hardware-based cache coherence

A hardware cache coherence protocol is commonly defined by associating, for each cache line, a **state** that represent the availability and use of that cache line inside the system. From a logical point of view, the state is global (i.e. the same for each cache of the architecture). A read/write operation to a cache line changes its state, and *may* prompt some communications between the caches to ensure that none of them holds a stale value. Most protocols rely on the concept of **invalidations**: when a line is modified on a private cache, *all the other caches* remove (if present) the old value. The critical point of this approach reside in efficiently maintaining the global state of each cache line, that can become difficult to implement and inefficient: in some notable cases the cost of accessing a cache line global state may end coinciding with the cost of accessing the memory, thus strongly limiting the beneficial effects of cache hierarchies.

Software-based cache coherence

The other side of the coin is to let the software manage cache coherency. Throughout the years several approaches were studied; an interesting survey can be found in [157]. It is important to note that, in general, a software-based protocol *may* still require some hardware facility, resulting in a mixed hardware-software approach. However, at least two important protocols can be completely implemented at software.

The initial idea, exploited in the first shared-memory multiprocessors, is quite simple: on private data and read-only shared data we do not really have the problem of cache coherence, as the values are either used by a single processor or immutable throughout the whole execution. Thus, we could use caches *only* for this kind of addresses, and *disable caching* on the others. This approach was exploited, for example, by the CMU C.mmp computer[186] in the '70s.

The idea can be generalized considering that a read/write shared memory area is usually modified by a single processor for a while; then, by means of synchronizations, all the other processor are informed of the completion of the operation and can start reading the new value. In this scenario, we can allow the caching of read/write shared memory: a processor may *hold* a cache line, to be the only allowed writer, and in the meantime the other processors cannot access the data; then, when the holder has finished working on the line, flushes its modifications from the cache to the memory and release the cache line, transforming it in read-only. This result in a flexible approach that only require the possibility of *flushing* a modified line back to the memory (for the writer) and *invalidating* a stale cache line (for the reader, to avoid finding an old value of the updated line in its cache). The use of these two operations can be delegated to the programmer, or automatically inserted by the compiler.

In the second case a compiler may not be able to correctly predict the access pattern and, to ensure correctness, end in producing a number of unnecessary flush and invalidation operations (up to the point that data is not really cached) that reduces the performance of the program[57, 157].

Scratchpad memory

A different approach to caches in general, proposed several years ago, is the use of **scratchpad memories**. The idea, introduced by Banakar and others [24] is quite simple and, in many ways, resemble the software cache coherence approach. Firstly introduced for power consumption problems, the rationale is to employ a fast static memory but *removing* all the complex mechanisms used in caches to access data by content. The idea is that the programmer (or the compiler) handle the scratchpad space by means of explicit memory transfer requests. Moving the most used data in

the scratchpad, a programmer can still exploit locality, with an approach similar to the one used with general purpose registers. In this scenario, of course, *coherence* can be easily handled because we explicitly know the content of each scratchpad, and thus can decide when data should be moved to the shared memory level to allow sharing. The operations required to "ensure" the correctness is still the flushing operation, to "update" the shared memory. Invalidation is not really required, and substituted with a specific transfer between the memory and the scratchpad, when needed. A notable example of multi-core that exploit a scratchpad memory is the IBM Cell [106, 118].

At the time of the introduction of multiprocessors, with the available technology, the research world came to the conclusion that hardware cache coherency protocols generally allowed better performances w.r.t software-defined protocols[138]. This was the result of several aspects:

- The wide use of simple interconnection networks, such as buses, that allowed a simple, cost-free global cache state implementation by snooping (i.e. each cache continuously listen to the bus to catch all the memory requests and change the state of owned lines accordingly)
- The study on generic, shared-memory programs in which the data exchange pattern on the shared areas was unknown and not predicted by the compiler in software-based approaches.

However, even at the time some researchers objected this conclusion, noting that with more complex interconnection networks the snooping approach was not really feasible, and at that point the implementation of the global state could strongly limit the performance of a parallel application. In these cases a software-based approach could outperform the automatic, hardware-based one [1]. Today the increase of cores per chip made the use of bus or similar interconnections ineffective *even inside the chip*. In this scenario, the conventional wisdom is that on-chip cache coherence will not scale to the large number of cores expected to be found on future processor chips, and thus new approaches must be taken[58, 79, 104].

In fact, this path has already been taken: some multi-core architectures are already available with disabled or controllable cache coherence mechanisms: we are talking of the IBM Cell processor [106, 118], with its scratchpad memories, the Intel Single-Chip Cloud Computer [59, 93] (SCC) that encapsulate 48 cores, each one with its private cache, but with no automatic cache coherence mechanisms, and the Tilera Tile*Pro*64 [160] that allow to finely control the cache coherence mechanism for each virtual memory page.

One of the problems of software-based cache coherence protocols was the inability of the compiler to correctly identify the sharing patterns, but we believe that a structured parallel programming approach should avoid that, as the sharing pattern is intrinsic in the definition of the paradigm.

Moreover, it should be noted that the problem of cache coherence is enormously mitigated if we use a message-passing programming model: in this case each process works *only* in its local memory area, and sharing is only explicitly governed by send/receive primitives that can easily keep the interesting memory areas coherent with no overheads at all.

For this reason, with this chapter we start approaching the problem of software cache coherence targeted at parallel patterns. Because of the limited platform availability, however, the study must be considered only a seminal work: as of today only the Tile*Pro*64 architecture that allow the finer control required to let the parallel program disable the automatic cache coherence mechanisms for specific memory areas, and compare the results with hardware-based approaches on the same architecture.

In the following section, we will firstly analyze the possible overhead of having automatic cache coherence in current multi-cores; then, we will study possible implementations of a farm and a data-parallel with stencil on the Tile*Pro*64 architecture.

7.1 The cost of automatic cache coherence

Despite all the claims, automatic cache coherence have a cost, even in snoopy-based architectures. An interesting study has been reported in [91] where, by means of specific benchmarks, the authors measured the latency of read operations in two current multi-core architectures (an Intel Nehalem and an AMD Shangai processor), showing that *in fact* load/store latency and bandwidth are affected by the cache coherence state of the line. We report their latency results in Table 7.1.

<table>
<tr><td colspan="2">Processor</td><td colspan="4">Shangai</td><td colspan="4">Nehalem</td></tr>
<tr><td>Source</td><td>State</td><td>L1</td><td>L2</td><td>L3</td><td>RAM</td><td>L1</td><td>L2</td><td>L3</td><td>RAM</td></tr>
<tr><td>Local</td><td>M/E/S</td><td>3</td><td>15</td><td rowspan="3">41</td><td rowspan="3">208</td><td>4</td><td>10</td><td rowspan="2">38</td><td rowspan="3">191</td></tr>
<tr><td rowspan="3">Core1 (on die)</td><td>Modified</td><td colspan="2">119</td><td>83</td><td>75</td></tr>
<tr><td>Exclusive</td><td colspan="2" rowspan="2">179-208</td><td colspan="3">65</td></tr>
<tr><td>Shared</td><td colspan="3">38</td></tr>
<tr><td rowspan="3">Core4 (other die)</td><td>Modified</td><td colspan="3" rowspan="2">224</td><td rowspan="3">319</td><td colspan="3">300-320</td><td rowspan="3">311</td></tr>
<tr><td>Exclusive</td><td colspan="2" rowspan="2">268-313</td><td colspan="3">186</td></tr>
<tr><td>Shared</td><td colspan="3">170</td></tr>
</table>

Table 7.1: Memory read latencies for core 0, depending on the cache line state, for two x86 processors. Times in clock cycles.

We can easily notice that, in general, the access of a cache line in the *shared* state cost as much as reading a private data; *modified* and *exclusive* states, on the other hand, increase the access latency. It should also be noted that in this measurements we always pay the cost of snooping that, although limited, could still be avoided in an incoherent approach.

We decided to adapt the benchmark to the Tile*Pro*64 architecture, so that we could have a first insight of how the latencies are affected by the DDC mechanism. Of course we expect an increased overhead, as we deal with complex interconnection networks and protocols: as already explained in Chapter 5, an incoherent requests goes directly from the cache to the memory; a coherent request, on the other hand, is sent to the *home node*, then to the memory. The results are reported in Table 7.2.

	L1	L2	Remote L2 (Home)	RAM
Incoherent	2	8	120	120
Coherent	2	8	40-70	160-204

Table 7.2: Memory read latencies for core 0, on the Tile*Pro*64 architecture, with or without cache coherence. Times in clock cycles.

In this case we have a limited amount of possibilities: if the DDC is enabled, because of the write-through mechanism between the L2 and the *home node*, a cache line (for the local L2) will always be either **shared** or **invalid**. The real difference is therefore only in having or not the DDC enabled. We can notice that the memory access time is increased from 120 up to 204 clock cycles. The latency depends on the distance between the home node and the issuing processor, but we are talking of an overhead that goes from 33% up to 70% **for each memory load**.

Of course, even disabling cache coherence have a cost: with DDC, if another cache have the required value, we pay from 40 to 70 clock cycles, depending on the distance; with an incoherent mode we have no way of knowing if another cache has the copy, and always go to the memory paying 120 clock cycles. In fact, the ability of cache-to-cache transfers is always presented as one of the many benefits of automatic cache coherence mechanisms; however, it should be noted that cache-to-cache transfers mostly depend on the program and the cache sizes: for example, if the working sets of the processor do not fit in cache, the probability of exploiting cache-to-cache transfers is very limited.

It should also be noted that, on the Tile*Pro*64, enabling the automatic cache coherence in fact lower the amount of L2 cache space available for each core, thus possibly further increasing performance losses.

7.2 Optimizing cache coherence for the farm pattern

In this section we analyze how to tune a farm implementation to better exploit the cache coherence mechanisms of the Tile*Pro*64. The results of this section have been already published in [44], as part of our effort in porting of the FastFlow framework on the Tile*Pro*64 architecture.

The farm pattern, as we already saw in the previous chapters, is particularly suitable for a message passing implementation. Inside the farm each worker is practically independent, as it only synchronize with the supporting processes to select the task on which compute the calculation, and to notify the completion of a task. For this reason, a message-passing implementation of the farm, even of a shared memory architecture such as a multi-core, is usually very efficient. This is a first clue, suggesting that a software-based cache coherent protocol could be effective on this kind of patterns.

The FastFlow framework is not, however, a pure message passing environment. It does support data sharing by means of *pointer passing*: send and receive operations are still used, but just to exchange ownership to memory area (by means of memory pointers).

In this scenario we can easily think of three different implementation of the farm, each one with a different use of the cache coherence mechanisms.

7.2.1 Automatic cache coherence with hashed home node

This represent the standard way of using the Tile*Pro*64 architecture: when data is allocated by using the operating system malloc/free operations, for each virtual memory pages the home node is selected by using an hash function, applied at the cache line granularity, so that the lines composing the page are uniformly distributed among all the caches: each L2 cache will work as home for $1/64$ of the page.

Being in a farm, each worker will use a different task, and thus different memory areas. Let us say that worker w_x is computing task t_y. We have that the number of sharers for each cache line of t_y is approximately 1 (only w_x is using it at the moment); yet, an uniform distribution of the home nodes means that, on average, the cache lines of t_y will be not homed on w_x. Given the strongly inclusive rule of the homing (Chapter 5), we have that each cache line currently allocated in w_x will also be allocated in another L2 cache. Apply this reasoning for each worker, and we can easily understand that, on average, we will be able to exploit only half of the cache of each core, negatively affecting the performance of the application.

7.2.2 Automatic cache coherence with fixed home node

An interesting way of avoiding the problems of the hashing mechanism, and still maintain the automatic cache coherence, is to *manually* select the home node *for the entire virtual memory page*. In general this means that, w.r.t the previous mechanism, all the cache lines of a specific memory page will be homed on the same L2 cache, defined by the programmer.

In the case of the farm we can easily identify, for each task, an ideal home node: the one that will compute the task. So, for each task, once we have selected the computing worker, we will set the home node correspondingly. This way we expect

better results, as we should limit the space overhead of the DDC mechanism, making the home node overlapping with the cache that is actually using the data.

It should be noted, however, that in the Tile*Pro*64, the selection of the home node cannot be changed once the memory page is allocated. This means that, in a pointer passing environment such as FastFlow, it is usually necessary to copy the task on a new memory area after the worker is elected, to select the proper homing node (thus voiding all the effect of pointer passing).

7.2.3 Disabling automatic cache coherence

The other solution is to completely disable cache coherence. If we are using a message passing implementation, all the work can be *hidden* in the send and receive functions; unfortunately with FastFlow we do not have a pure message passing environment. However, we can still start from the implementation (where sharing points are well defined) to easily find the "hot spots", in which cache coherence must be handled; in particular

1. When the emitter has selected the destination worker for a specific task: any cache line related to the task in the cache of the emitter must be *flushed* back to memory, so that the worker will be able to gather the correct values.

2. When the worker starts a computation on a new task: any cache line related to the task must be *invalidated*, because the cache of the worker may have stale values for some of these lines from previous computations.

3. When the worker finishes a computation on a task: the result must be correctly notified to the collector, again, by means of *flush* operations on the cache lines related to the result.

4. Finally, when the collector want to receive a result from one of the workers, all the lines related to the result must be *invalidated* on the collector cache, again to avoid stale data.

As previously said, FastFlow is not a message passing environment, yet we can identify the four points where pointer send and receive happens, and *manually* apply flush and invalidate operations on the code.

It should be noted that this approach still represent a form of *compiler-driven* cache coherence, as the parallel code of the farm is not written by the programmer, but by our compiler (or by the library developer in the case of FastFlow). So, it is not the application programmer that is adding cache coherence operations to its code.

7.3 Experimental Results

We tested the three implementation using a farm executing the Matrix Multiplication algorithm. This is an interesting kernel to test cache hierarchies because, as long as matrices are small enough to stay in cache, data reuse is very high: $O(N^3)$ operations are performed over $O(N^2)$ memory transfers. When matrices exceed cache sizes, the number of transfers grows to $O(N^3)$, in fact executing one operation per element. Given the results of the previous chapters we expect, in the latter, that the memory system will become a bottleneck. We identified in this the perfect example to test the different cache coherence oriented implementations to check if and how much a specific cache-coherence policy affects the performance of the application.

We wrote a FastFlow application exploiting the farm skeleton on a stream of 3200 matrices. For this kind of experiments we selected an **SMP-like** memory allocation policy, as FastFlow, with its pointer passing mechanism does not allow to transparently implement the **NUMA-like** family of allocation policies.

Small matrices

We started with matrices of 64×64 integer elements. The results are reported in Table 7.3 and Figure 7.1. This represent a very interesting example of the space overhead of DDC.

Each matrix takes 16KB of space, so that the entire working set of each worker for each task is 48KB (two input matrices plus an output one). This means that the working set is small enough to fit in the L2 cache of one tile and therefore the number of memory transfers are minimized. We are expecting an extremely high scalability, that is in fact verified with the fixed home and incoherent implementations. Instead, the standard cache coherency protocol works very well up to $\sim$ 20 nodes, then suddenly stops scaling. This is because of the L2 halving effects previously described: with a large parallelism degree, the L2 cache available for each tile is less than the required 48KB. In this case, the working set of the algorithm does not fit in the cache and the performance *of the sequential code* executed by the workers suddenly decrease. The result is pretty atypical for a farm pattern, in which we usually reach a point in which the pattern stop to scale, but the performance tend to not decrease by adding more workers.

This shows that does not exist, even in hardware cache coherency protocol, an implementation that fits all the cases: with this algorithm the use of a hashed home is really bad, while just by manually fixing the home on the specific worker we are able to reach an almost ideal speedup.

On the other hand the incoherent implementation is indeed very good, as it is able to obtain aligned results with the best option for automatic cache coherence. There is not really much to say except that a software-based approach for this farm

Par. Degree	Completion Time			Speedup		
	Coherent		Incoherent	Coherent		Incoherent
	Hashed	Fixed		Hashed	Fixed	
1	14914.31	15051.23	15059.93	1.01	1.00	1.00
2	7469.24	8042.82	7537.49	2.02	1.87	2.00
3	4989.50	5157.98	5027.72	3.02	2.92	3.00
4	3744.65	3926.27	3771.24	4.02	3.84	3.99
5	3001.07	3141.46	3017.11	5.02	4.79	4.99
6	2511.71	2620.16	2517.26	6.00	5.75	5.98
7	2156.59	2252.99	2159.78	6.98	6.69	6.97
8	1891.29	1971.66	1889.20	7.96	7.64	7.97
9	1686.90	1756.56	1680.37	8.93	8.57	8.96
10	1525.30	1578.55	1515.61	9.87	9.54	9.94
11	1386.88	1439.46	1376.10	10.86	10.46	10.94
12	1274.65	1319.38	1261.72	11.82	11.42	11.94
13	1181.85	1218.83	1165.73	12.74	12.36	12.92
14	1104.20	1130.99	1082.33	13.64	13.32	13.92
15	1032.75	1057.63	1010.53	14.58	14.24	14.90
16	1005.66	993.98	949.49	14.98	15.15	15.86
17	916.02	939.20	892.95	16.44	16.04	16.87
18	869.88	886.61	843.87	17.31	16.99	17.85
19	828.16	838.27	799.02	18.19	17.97	18.85
20	789.47	798.35	760.46	19.08	18.87	19.81
22	1230.46	726.24	691.07	12.24	20.74	21.79
24	1525.99	669.26	639.36	9.87	22.50	23.56
26	1538.54	613.18	586.73	9.79	24.56	25.67
28	1486.31	572.16	547.84	10.13	26.32	27.49
30	1517.25	542.00	510.11	9.93	27.79	29.53
32	1408.96	509.53	481.80	10.69	29.56	31.26
34	1450.11	481.71	462.15	10.39	31.27	32.59
36	1344.83	449.19	441.80	11.20	33.53	34.09
38	1437.04	431.75	405.46	10.48	34.88	37.15
40	1374.87	404.91	394.42	10.95	37.20	38.19
42	1328.20	388.17	374.30	11.34	38.80	40.24
44	1822.93	368.54	376.76	8.26	40.87	39.98
46	1294.54	358.26	340.25	11.63	42.04	44.27
48	1216.66	340.75	340.92	12.38	44.20	44.18
50	1199.77	324.73	312.88	12.55	46.38	48.14
52	1182.05	315.84	313.35	12.74	47.69	48.07
54	1474.749	303.72	294.79	10.21	49.59	51.09

Table 7.3: Comparison between cache coherence methods for the matrix multiplication, 64×64 elements. Completion Times in milliseconds.

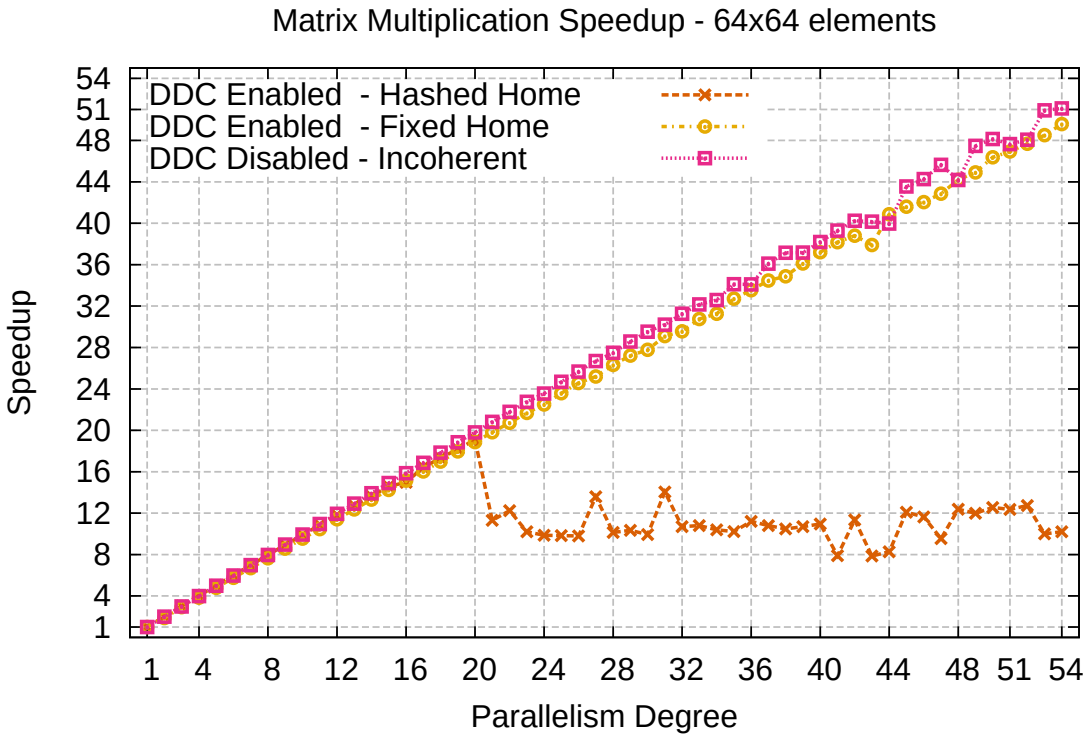


Figure 7.1: Comparison between cache coherence methods for the matrix multiplication, 64×64 elements.

works as good as a manually tuned automatic approach, and much better than the default one.

Large matrices

By using larger matrices we expect the working set to not fit the caches in any of the policies. Still, it represent an interesting experiment as we are stressing the memory, and thus we can actually see if the coherency protocol helps or aggravate the situation. We use 128×128 matrices, resulting in a working set of 192KB, far larger than the 64KB of L2 caches. The results, shown in Table 7.4 and Figure 7.2, highlight some interesting facts.

First of all, it is notable that the version with a single worker is actually much faster when using the standard cache coherence policy. The reason is to be found in the values of Table 7.2: when the home is uniformly distributed, we have that 192KB is, of course, larger than the single cache; yet, the homing nodes have sufficient space to hold the entire working set remotely. This means that we are in one of that cases in which automatic cache coherence allow cache-to-cache transfers, that are indeed faster than memory transfers. Of course, fixing the home node to *a single cache* does not exhibit this side effect, as the home coincide with the local node. For this reason, we observe low speedups with a limited parallelism degree when using the

Par. Degree	Completion Time			Speedup		
	Coherent		Incoherent	Coherent		Incoherent
	Hashed	Fixed		Hashed	Fixed	
1	132784.78	155219.78	151816.12	1.02	0.87	0.89
2	77401.46	79135.46	76792.41	1.75	1.71	1.76
3	56555.57	54111.38	52874.43	2.39	2.50	2.56
4	44322.29	41502.51	40199.35	3.05	3.26	3.37
5	37458.63	34120.08	31895.36	3.61	3.97	4.24
6	31852.85	29606.95	26705.87	4.25	4.57	5.07
7	28013.07	25071.60	23240.65	4.83	5.40	5.83
8	25036.38	22935.86	20364.07	5.41	5.90	6.65
9	22970.17	20875.99	18220.54	5.89	6.49	7.43
10	21195.02	18928.68	16538.96	6.39	7.15	8.19
11	19775.85	17489.82	15121.06	6.85	7.74	8.95
12	18484.45	16407.90	13863.40	7.32	8.25	9.77
13	17512.43	15523.90	13160.40	7.73	8.72	10.29
14	16749.38	14757.21	11941.64	8.08	9.17	11.34
15	15911.66	13993.66	11288.20	8.51	9.68	11.99
16	15194.49	13358.30	10571.47	8.91	10.14	12.81
17	14647.91	12858.15	9965.87	9.24	10.53	13.59
18	14133.48	12189.85	9455.58	9.58	11.11	14.32
19	13709.81	11788.70	9028.33	9.88	11.49	15.00
20	13249.47	11532.12	8584.54	10.22	11.74	15.77
22	12790.31	10894.88	7823.57	10.59	12.43	17.31
24	12084.48	10613.17	7244.21	11.20	12.76	18.69
26	11529.80	9938.42	6734.66	11.74	13.62	20.10
28	11097.92	9515.11	6268.21	12.20	14.23	21.60
30	10814.82	9185.34	5860.50	12.52	14.74	23.10
32	10876.37	8755.90	5554.63	12.45	15.46	24.37
34	10330.55	8435.27	5247.86	13.11	16.05	25.80
36	10211.39	8077.54	5021.73	13.26	16.76	26.96
38	9949.63	7712.97	4754.31	13.61	17.55	28.48
40	9794.56	7407.29	4527.12	13.82	18.28	29.91
42	9682.01	7054.67	4333.57	13.98	19.19	31.24
44	10258.91	6823.47	4157.72	13.20	19.84	32.56
46	9459.94	6527.04	3999.12	14.31	20.74	33.86
48	9756.67	6352.31	4007.70	13.88	21.31	33.78
50	10217.60	6168.42	3717.81	13.25	21.95	36.42
52	10080.67	5950.94	3677.54	13.43	22.75	36.82
54	10111.63	5756.50	3640.95	13.39	23.52	37.19

Table 7.4: Comparison between cache coherence methods for the matrix multiplication, 128×128 elements. Completion Times in milliseconds.

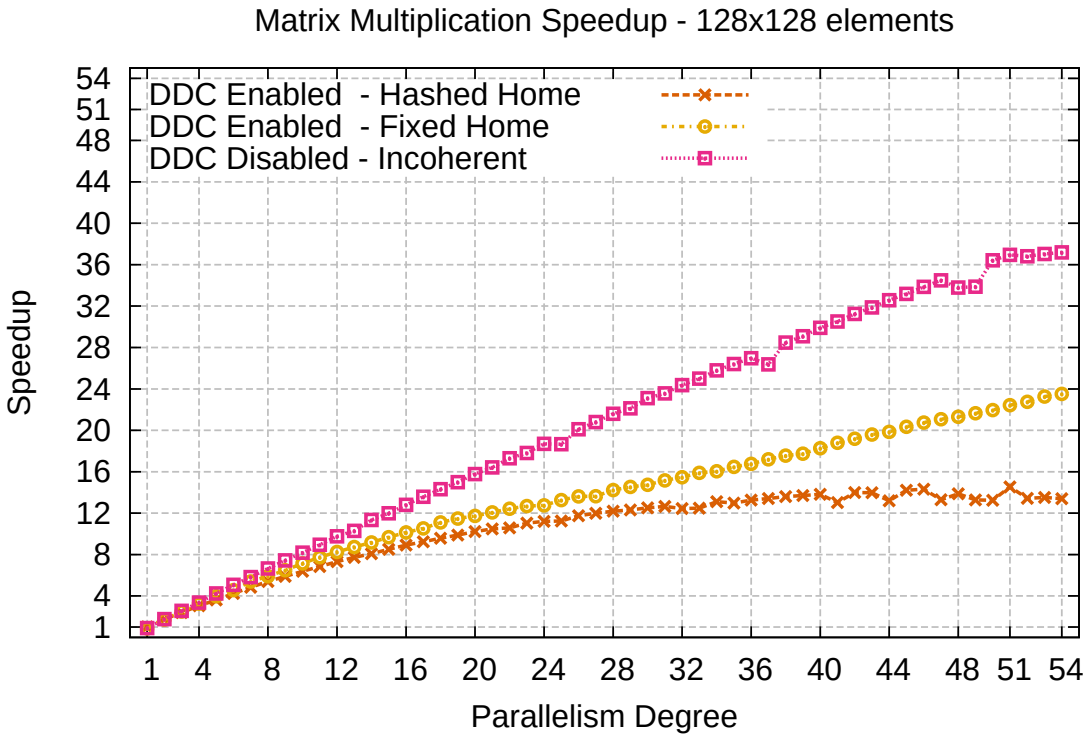


Figure 7.2: Comparison between cache coherence methods for the matrix multiplication, 128×128 elements.

two non-default policies: the sequential code has become slower.

Nevertheless, the hashing is not working well in parallel: starting with a parallelism degree of 2 the speedup of the incoherent version is already better, and with 3 the default implementation has been overtaken even by the fixed home implementation. This represent a further example of the limited applicability of cache-to-cache transfers: by just having more than two workers, their effect is completely vanished. Of course this also depend on the parallel pattern: the farm, in which each worker is independent, does not offer many possibilities for cache-to-cache transfers.

The incoherent policy works surprisingly good: by removing the cache coherency protocol we probably reduce the amount of memory requests, or at least the amount of traffic on the mesh, ending with far better results w.r.t any implementation that exploit automatic cache coherence.

7.4 Optimizing cache coherence for a data-parallel pattern

Given the promising results of software-managed cache coherence on the Tile*Pro*64 architecture, we decided to try exploiting the same concepts on a data-parallel with

stencil pattern. We took the data-parallel FFT implementation presented in Chapter 6 and adapted it to exploit the previously defined approaches. Given the sharing pattern we also tried another use of the mechanisms offered on the Tile*Pro*64, by disabling the *local* L2 cache and using the group of caches only as a global distributed cache.

7.4.1 Automatic cache coherence with hashed home node

This represent the standard way of using the Tile*Pro*64 architecture. The idea is very simple: at the end of each iteration a global barrier is performed, the source and destination buffers are exchanged and the following iteration can start. The pseudocode is reported in Listing 7.1; further details are given in Chapter 6, Section 6.5.1.

```
Worker i:
  for (q=0; q<log2(N); q++) {

    Compute First Partial Block
    Compute Complete Blocks
    Compute Last Partial Block
    Barrier
    Src-Dst Buffer Exchange

  }
```

Listing 7.1: Shared Memory FFT Worker.

It is important to notice that, like with the previous pattern, the local partition of each worker can be modified only by the worker itself but will be read by other workers in the following iteration. Even in this case, however, the use of a hashing mechanism will probably end in limiting the amount of L2 space available for the computation.

7.4.2 Automatic cache coherence with fixed home node

Even in this application we can try to avoid the problems of the hashing mechanism by fixing the home node. In this case, given the *owner compute rule*, we can identify, for each worker, its *local partition*, and fix the home node on the proper workers. It is interesting to note that the selection of the local partition happen with the same way in which, in the previous chapter, we selected the memory controller.

In this case the fact that the selection of the home node cannot be changed once the memory page is allocated is not really a problem, as we are not working on streams, the memory is allocated *before* the computation and the partitioning is not done dynamically, so we just need to ask the programmer to use a properly allocated area to fill the input data of the parallel pattern.

7.4.3 Disabling local caches

In this pattern, even by using the fixed node implementation, we usually end in having multiple copies of a cache line in the system: when a worker is reading another partition, the line will have to stay in the owner node, that works as home, and in the local node that is reading the line. This may lead to some inefficiency, as the owner node may not be interested in using the line in that particular moment. An interesting approach to try to solve the problem is by *disabling the local L2 caches*. This way we still keep the memory cacheable (at the L1 level and at the DDC level) but we reserve all the space of the L2 caches for working as home nodes. This ensure the existence of a single copy of a cache line in the whole set of L2 caches; the problem is that now, on a L1 miss, the request *may* need to reach a remote node, possibly distant, increasing the cost of L2 misses. Having the home nodes enabled, we ensure automatic cache coherence, so no particular modifications must be made to the code.

7.4.4 Disabling automatic cache coherence

Finally, we study a way of handling cache coherence by means of flush and invalidate instructions even in this data-parallel pattern.

Let us assume, for the moment, that we are at iteration i, all the caches are empty and that the results of iteration $i-1$ have been written in memory. In this situation each worker may work independently with its cache, handling everything as private data: it will read values of other partitions, but belonging to the previous iteration and thus already saved in memory, and write values of its own partition that, however, will not be read until the next iteration. At the end of its iteration, *before* entering the barrier, the worker can simply flush all its modified lines to the memory, so that at the beginning of the next iteration the correct values are stored in memory. Because of the owner compute rule, the modified lines will only belong to its own partition, so the rest of the data is not affected by this operation. The worker should also ensure that, at the next iteration, no stale data belonging to other partitions will be found on its cache. A conservative possibility is to *invalidate* all the data belonging to the other partitions; this simplistic solution is reported in Listing 7.2, where we show the high-level Tilera library call used for the task; these are translated in a sequence of assembler instruction, one for each cache line belonging to the requested area. However, in theory, if we know the stencil we can just invalidate the data really needed in the next iteration. After these operations, the worker can finally enter the barrier. When all the workers have completed the iteration, we are in the same condition of the beginning of the iteration: caches does not contain the remote partitions and the memory contains an updated version of the whole array. We can safely start the next iteration.

It should be noted that all of this is made possible by exploiting the *owner compute rule*, that ensure that each partition (and thus cache line) is modified by a

```
Worker i:
  for (q=0; q<log2(N); q++) {
    Compute First Partial Block
    Compute Complete Blocks
    Compute Last Partial Block
    foreach partition p {
      if isOwned(p)
        tmc_mem_flush(p, sizeof(p))
      else
        tmc_mem_inv(p, sizeof(p))
    }
    Barrier
    Src-Dst Buffer Exchange
  }
```

Listing 7.2: Shared Memory FFT Worker with per-partition invalidations.

```
Worker i:
  for (q=0; q<log2(N); q++) {
    Compute First Partial Block
    Compute Complete Blocks
    Compute Last Partial Block


    tmc_mem_flush_l2()




    Barrier
    Src-Dst Buffer Exchange
  }
```

Listing 7.3: Shared Memory FFT Worker with L2 cache flushing.

single worker; otherwise, we would end in writing partial cache lines in the memory, undermining the correctness of the algorithm.

Finally, in our implementation, we tested several ways of flushing and invalidating the caches. The Tile*Pro*64 allow selective invalidation and flush of single cache lines; however, in our experiments, if the data structures are far bigger than the caches, it becomes more convenient to completely empty the local L2 cache than issuing an invalidation/flush for every line that could possibly be in the cache; this last solution is exemplified in Listing 7.3, again by using the corresponding Tilera library call. Of course, we note again that these instructions are not really introduced by the programmer, but by the compiler that produces the parallel implementation: again, the use of incoherent memory areas is still transparent to the user.

7.5 Experimental Results

We executed the experiments using the same configuration presented on Chapter 6. The FFT is calculated over an array of 1048576 fixed-point elements. In this case we present the results using the two most effective memory allocation polices: **NUMA-like, RR** and **SMP-like, linear**, to also show whether or not the two different optimizations (memory allocation and software cache coherence) can be mixed together, and the possible outcomes. In fact, for the **SMP-like, linear** allocation, given the interleaved memory allocation, we decided to avoid a fixed homing selection, that seem an unnatural choice. All the other combinations are presented in Figure 7.3 and Tables 7.5 and 7.6, showing really promising results for the incoherent implementation, that is actually able to overcome the other policies.

Thus, even in this case, the overhead of invalidating and flushing data is to be considered negligible. The final results shows, with the NUMA-like allocation, a

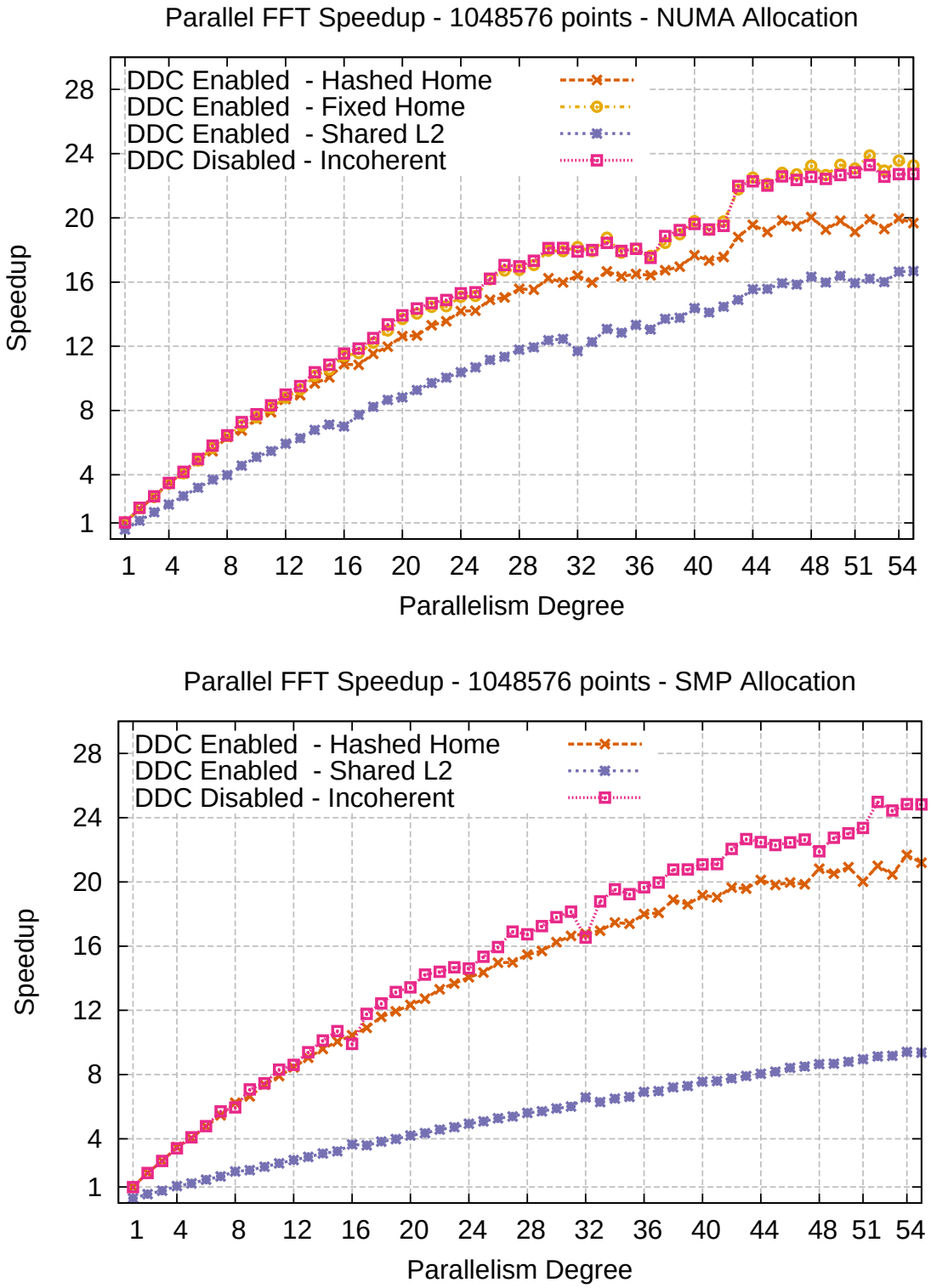


Figure 7.3: Comparison between cache coherence methods for the FFT, 1048576 points.

Par. Deg.	Completion Time				Speedup			
	Coherent			Inc.	Coherent			Inc.
	Hashed	Fixed	Shared		Hash	Fix	Share	
1	724.57M	701.30M	1216.23M	704.01M	1.00	1.03	0.60	1.03
2	387.29M	376.01M	633.95M	373.17M	1.87	1.93	1.14	1.94
3	276.23M	277.64M	435.29M	272.29M	2.62	2.61	1.66	2.66
4	206.51M	209.55M	336.50M	207.64M	3.51	3.46	2.15	3.49
5	177.79M	177.14M	270.29M	173.14M	4.08	4.09	2.68	4.18
6	149.08M	148.05M	227.03M	145.37M	4.86	4.89	3.19	4.98
7	132.63M	127.87M	196.07M	124.67M	5.46	5.67	3.70	5.81
8	114.00M	113.13M	181.87M	112.17M	6.36	6.40	3.98	6.46
9	107.32M	102.66M	158.62M	99.52M	6.75	7.06	4.57	7.28
10	97.01M	95.64M	142.09M	93.25M	7.47	7.58	5.10	7.77
12	83.11M	82.24M	122.05M	80.58M	8.72	8.81	5.94	8.99
14	74.81M	71.30M	106.74M	69.76M	9.69	10.16	6.79	10.39
16	66.60M	63.53M	103.35M	62.73M	10.88	11.40	7.01	11.55
18	62.83M	59.25M	87.96M	57.92M	11.53	12.23	8.24	12.51
20	57.36M	52.89M	82.14M	52.01M	12.63	13.70	8.82	13.93
22	54.44M	50.12M	74.59M	49.31M	13.31	14.46	9.71	14.69
24	51.07M	48.06M	69.76M	47.37M	14.19	15.08	10.39	15.30
26	48.64M	44.81M	64.99M	44.71M	14.90	16.17	11.15	16.21
28	46.52M	43.24M	61.45M	42.68M	15.58	16.76	11.79	16.98
30	44.63M	40.34M	58.56M	40.00M	16.23	17.96	12.37	18.12
32	44.17M	39.85M	61.93M	40.47M	16.40	18.18	11.70	17.90
34	43.48M	38.61M	55.38M	39.29M	16.66	18.77	13.08	18.44
36	43.91M	40.12M	54.35M	40.09M	16.50	18.06	13.33	18.07
38	43.29M	39.29M	52.86M	38.40M	16.74	18.44	13.71	18.87
40	41.00M	36.61M	50.39M	36.93M	17.67	19.79	14.38	19.62
42	41.25M	36.65M	50.07M	37.16M	17.56	19.77	14.47	19.50
44	37.04M	32.19M	46.59M	32.52M	19.56	22.51	15.55	22.28
46	36.53M	31.77M	45.48M	32.09M	19.84	22.80	15.93	22.58
48	36.15M	31.18M	44.36M	32.13M	20.04	23.24	16.33	22.55
50	36.56M	31.09M	44.22M	31.97M	19.82	23.30	16.38	22.66
52	36.39M	30.33M	44.71M	31.11M	19.91	23.89	16.20	23.29
54	36.31M	30.75M	43.52M	31.89M	19.96	23.56	16.65	22.72
56	35.97M	29.17M	42.01M	30.98M	20.14	24.84	17.25	23.39
58	36.32M	30.30M	42.08M	30.64M	19.95	23.91	17.22	23.65
60	36.10M	29.63M	42.00M	30.52M	20.07	24.46	17.25	23.74
62	36.81M	30.11M	41.42M	32.01M	19.68	24.07	17.49	22.64
63	36.95M	30.44M	42.50M	32.59M	19.61	23.80	17.05	22.23

Table 7.5: Comparison between cache coherence methods for the FFT, 1048576 points. NUMA-like allocation policy. Completion Times in clock cycles.

Par. Degree	Completion Time			Speedup		
	Coherent		Incoherent	Coherent		Incoherent
	Hashed	Shared		Hashed	Shared	
1	722.05M	2448.75M	725.48M	1.00	0.29	1.00
2	382.37M	1295.87M	386.48M	1.89	0.56	1.87
3	276.79M	943.83M	275.35M	2.61	0.77	2.62
4	207.05M	686.42M	211.97M	3.49	1.05	3.41
5	178.86M	592.61M	176.21M	4.04	1.22	4.10
6	151.39M	497.78M	150.93M	4.77	1.45	4.78
7	132.15M	435.31M	126.52M	5.46	1.66	5.71
8	115.74M	368.45M	121.22M	6.24	1.96	5.96
9	108.78M	352.16M	102.01M	6.64	2.05	7.08
10	97.97M	319.18M	96.83M	7.37	2.26	7.46
12	85.50M	269.74M	83.89M	8.45	2.68	8.61
14	75.25M	234.06M	71.36M	9.60	3.08	10.12
16	69.08M	198.02M	72.89M	10.45	3.65	9.91
18	62.26M	188.83M	58.07M	11.60	3.82	12.43
20	58.52M	171.85M	53.76M	12.34	4.20	13.43
22	54.24M	157.78M	50.14M	13.31	4.58	14.40
24	51.35M	146.35M	49.45M	14.06	4.93	14.60
26	48.25M	136.73M	45.31M	14.97	5.28	15.94
28	46.68M	128.57M	43.14M	15.47	5.62	16.74
30	44.45M	122.57M	40.56M	16.24	5.89	17.80
32	43.04M	109.75M	43.67M	16.78	6.58	16.53
34	41.34M	111.03M	36.96M	17.47	6.50	19.54
36	40.11M	104.37M	36.73M	18.00	6.92	19.66
38	38.24M	100.27M	34.78M	18.88	7.20	20.76
40	37.66M	95.51M	34.23M	19.17	7.56	21.09
42	36.76M	93.07M	32.75M	19.64	7.76	22.05
44	35.89M	89.72M	32.12M	20.12	8.05	22.48
46	36.18M	85.72M	32.14M	19.96	8.42	22.46
48	34.68M	83.44M	32.97M	20.82	8.65	21.90
50	34.52M	82.07M	31.36M	20.92	8.80	23.02
52	34.39M	79.04M	28.90M	21.00	9.14	24.98
54	33.31M	76.75M	29.06M	21.67	9.41	24.85
56	32.61M	74.66M	29.42M	22.14	9.67	24.54
58	33.95M	73.94M	28.91M	21.27	9.77	24.98
60	31.81M	71.17M	28.43M	22.70	10.15	25.40
62	33.07M	68.71M	28.74M	21.83	10.51	25.13
63	32.20M	70.35M	29.16M	22.42	10.26	24.76

Table 7.6: Comparison between cache coherence methods for the FFT, 1048576 points. SMP-like allocation policy. Completion Times in clock cycles.

scalability of $\sim$ 23 for the incoherent implementation, compared to $\sim$ 19 of the default one and of $\sim$ 17 obtained by disabling the local caches. With SMP-like allocation we have similar results.

It is interesting to notice that, again, the fixed home implementation works pretty well, resulting in the same performance of the incoherent one: this approach should be better studied, as it seems that we are able to achieve the improvements of the incoherent mechanism without the complexity of manually inserting invalidation and flush operations.

Finally, clearly the idea of disabling the local caches, although interesting, is not really offering good results: the sequential code is more than three times slower in the SMP implementation, meaning that we really pay too much the absence of a local L2 cache: the first level of cache alone is not sufficient to guarantee a reasonable amount of locality.

7.6 Summary

In this chapter we introduced the reader to the problem of automatic cache coherence, and its cost in current and future multi-core architectures, where the use of a complex interconnection network introduces major problems in effective implementations of hardware-based mechanisms. In this scenario, we expect the emergence of multi-core architectures that, similarly to the Tile*Pro*64, will allow the programmer to configure the automatic cache coherence to better fit the program. In this scenario, the possibility of working *without* automatic mechanisms must be taken into account. The use of parallel patterns allow the introduction of software-based protocols *only* in the support level, making the transition transparent to the user. With this chapter we presented some preliminary results, implementing two of the most used parallel patterns: a farm and a data-parallel with stencil. In our experiments, disabling cache coherence really offer performance improvements, motivating the need of a deeper analysis of this approach.

Part IV

Wrapping Up

Chapter 8

Wrapping up: compiling a parallel module on Tile*Pro*64

With this final chapter we put together the results obtained throughout the thesis, by taking an example application and its graph of parallel modules. We will focus our study on a single module, defining possible implementations and evaluating them with the performance model of the previous chapters to estimate the service time function $T_s^{(n)}$ on the Tile*Pro*64 architecture. The result of the study will be compared with the times gathered by running the implementations on the architecture, to analyze the accuracy of our predictions and, therefore, the degree of fidelity of the hypothetical parallel compiler in:

- predicting the service time of the parallel version (and thus the parallelism degree required to sustain the input throughput);
- identifying the best implementation for the selected parallelism degree.

8.1 Example module and its application

For the sake of simplicity, clarity and conciseness, we will reuse some of the studies from the previous chapters to limit, as much as possible, the introduction of new concepts and modeling ideas.

For this reason we decided to use one of the algorithms already examined in Chapter 6, so that the reader is already familiar with the sequential problem and the expected performance results in term of both the sequential and the parallel version. Among the set, the **Sobel Operator** seemed quite interesting, as it is more complex than simple array operators and its application area (i.e. video analysis) makes it, by definition, a stream-oriented operator. Because of this, it indeed make sense to treat it as a module of a parallel application that works with streams.

We imagined a video surveillance application, in which the Tile*Pro*64 (together with other parallel machines, perhaps heterogeneous) receive a stream of multiple

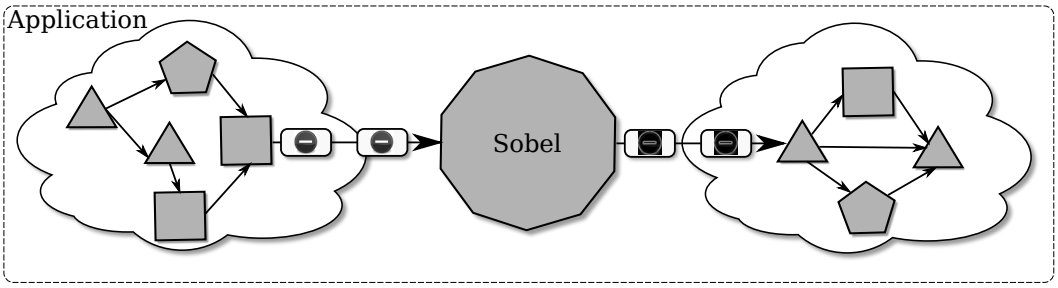


Figure 8.1: Example application graph, with particular focus on the Sobel module.

compressed videos from the network interface and perform a feature extraction algorithm to identify moving targets. An example of such application is `bodytrack`, contained in the **PARSEC**[38] benchmark suite. One of the first steps of this kind of algorithms is to perform an edge detection to identify objects. Our **Sobel Operator** is, as such, a good candidate for representing a module of this application. Without going in further details, we expect a graph composed of several parallel modules in which, *at some point*, we have the edge detection phase, executed by our **Sobel Module**, as depicted graphically in Figure 8.1.

In particular, we expect the **Sobel Module** to receive pre-processed images, already in the greyscale uncompressed format, so that the module is *only* in charge of applying the Sobel operator to a stream of images. The result will be, of course, forwarded to the following modules in the application graph.

It is important to notice that, w.r.t the approach of Chapter 4, we are not trying to *deploy* the application, so our intent here is not to find the parallelism degree of each module to obtain the best service times of the whole application. We are making a step backward, focusing on a single module and trying to *estimate* the service time function $T_s^{(n)}$ of the module. This is, of course, just a part of the compilation phase in which we parallelize each module to remove the bottlenecks. Yet, measuring the accuracy of the single module performance model is a necessary step towards the realization of the programming environment we aim to, as described in Chapters 3 and 4.

In this scenario we study the parallel module *isolated*, assuming the inter-arrival time $t_A \simeq 0$, so that the module is always a bottleneck, and thus its service time is never limited by the inter-arrival time of tasks. This is important to study the behavior of different parallel implementations of the module in a general case.

To keep the study realistic, we imagine that a portion of the Tile*Pro*64 will be used to execute the other parts of the application or, at least, handle the input/output streams towards other modules if they reside on other architectures. Our study will therefore use $^3/_4$ of the cores and 3 memory controllers, as depicted in Figure 8.2, for the Sobel Module. This way we can safely assume that other parts of the application will be allocated on the Tile*Pro*64, but will not affect the performance of our module,

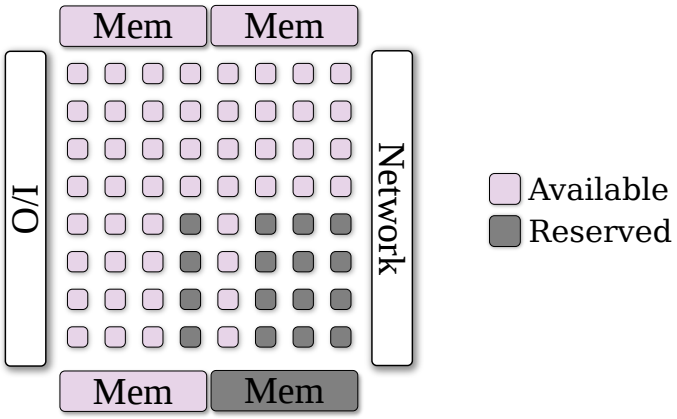


Figure 8.2: Part of the Tile*Pro*64 architecture available for the module.

as they can work with a separate memory interface.

8.2 Parallel pattern and its implementations

Before continuing, we briefly review the algorithm and its data dependencies to motivate the possible patterns.

The **Sobel operator** is one of the most simple and common gradient estimator, able to compute an approximate value for the gradient in each image point in a very simple yet effective way. Specifically[182], for each pixel of the image the operator approximate the derivatives $\frac{\delta f}{\delta x}$ and $\frac{\delta f}{\delta y}$ by differences, considering a 3×3 neighboring of the point in the following way:

$$\frac{\delta f}{\delta y} = (x_6 + 2x_7 + x_8) - (x_0 + 2x_1 + x_2)$$
$$\frac{\delta f}{\delta x} = (x_2 + 2x_5 + x_8) - (x_0 + 2x_3 + x_6)$$

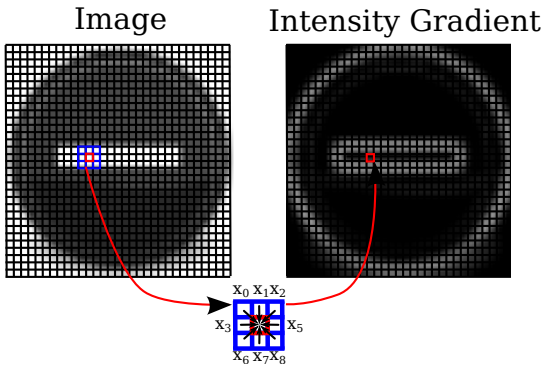


Figure 8.3: Sobel Operator.

The complete calculation for each pixel therefore consist in 4 multiplications, 15 sums and 2 absolute values. The operation and the required dependencies among pixels is graphically exemplified in Figure 8.3. This is executed once for each pixel of the image, resulting in a linear memory access pattern.

8.2.1 Parallel Patterns

Given the stream-based environment, where the operator must be applied to several images, we can approach the problem using a stream-parallel pattern such as the **farm**.

We can also parallelize the module using a data-pattern pattern, that with this algorithm is quite interesting: at a first look one may think to a **data-parallel with stencil**, as each point require all the neighbors. Yet, a deeper study reveal that the stencil is required on read-only data (i.e. the input image), so we can rewrite the stencil by introducing a slight amount of data replication and obtain a **map** pattern, following the very same approach described in Chapter 3, Section 3.2.4.

For the sake of simplicity we will study the **farm**, because we already introduced a performance model for this pattern in Chapter 4, Section 4.2. The preliminary model will be extended when needed during the study of the different implementations.

8.2.2 Farm Implementations

The farm pattern can be considered a cornerstone for high level parallel programming, as it covers one of the most important parallelization method of stream-based applications.

Despite the ubiquity of this parallel paradigm, that is available in most structured parallel programming tools, there is a surprisingly scarce amount of work on efficient farm implementations for multi-core. During the golden age of skeleton programming a lot of work focused on this topic, but at that time parallel machines were mainly distributed memory architectures; in the last decade, with the emergence of multi-cores, we saw a shift towards shared memory architectures that, however, remained uncovered in these works.

Finally, it is worth noting that most of the widely used high-level parallel libraries and languages targeted at multi-core or shared memory, like OpenMP[56] and Intel Threading Building Blocks[144] are mainly data-parallel environments, and therefore their support for stream-based programs is very limited. In fact, both support data-parallel implementations using farms (i.e. for maps with dynamic data partitioning and divide-and-conquer skeletons), but the *streaming* behavior is missing, so we cannot easily build an OpenMP or TBB farm working on streams.

We will define three different implementations, each of them derived by refinement of the previous, starting with the one presented in Chapter 4, and ending with the implementation used in **FastFlow**[12], that can be probably considered the current state of the art for farm implementations on multi-cores.

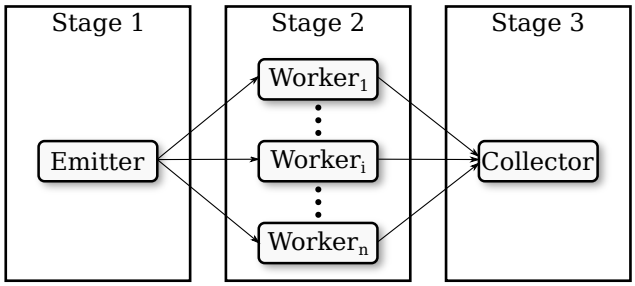


Figure 8.4: Emitter-Worker-Collector scheme for the farm pattern.

Initial implementation: Message Passing

We start with the farm scheme already discussed in Chapter 4, Section 4.2. The farm is composed of an **Emitter** to distribute tasks, multiple **Workers** to compute the Sobel operator and a **Collector** to gather the results. The scheme is depicted in Figure 8.4. The program uses a local environment and exploit message passing to exchange data. The use of a local environment is interesting, as it allow us to:

- Seamlessly inherit the performance model already discussed in Chapter 4.
- Define an implementation that uses incoherent memory areas, as each processor has its local memory and exchange data by means of send/receive functions, that can be easily modified to manually handle the cache coherence for the messages, by explicitly flushing data back to the memory. This approach allow us to use the hardware cost model presented in Chapter 5, and should provide the performance benefits shown in Chapter 7.
- Adopt the **NUMA-like RR** allocation policy for the workers: when the worker for a specific task is selected by the emitter, this can copy the message in a memory area explicitly allocated on the memory controller corresponding to the worker. This allocation policy should provide better performances, as shown in Chapter 6.

To further improve the performance, we adopt zero-copy communications, where data transfer requires no additional copy operations neither at the sender nor at the receiver. On traditional message-passing systems like distributed-memory architectures, zero-copy communications have been implemented relying on advanced networking infrastructures and devices (e.g. InfiniBand and Myrinet). On shared-memory architectures, zero-copy can be implemented by using in-memory copy operations and relying entirely on user-space operations. This idea is considered a state-of-the-art technique already applied in several research works in the literature, such as [99].

Zero-copy allow us to implement a send-receive with a single message copy, instead of the two usually required. This already represents a first improvement w.r.t

the send/receive primitives presented in Chapter 4, Section 4.2.5.2, and even common shared memory implementations of MPI). To summarize, we expect the following service times for each process (assuming the copy is executed by the sender):

$$\begin{aligned} T_{S-E} &= T_{copy} \\ T_{S-W_i} &= T_{W-calc} + T_{copy} \\ T_{S-C} &= \sim 0 \end{aligned}$$

Overlapping communications: Message Passing with Copy on Receive

Although there exist several MPI implementations for shared memory architectures, using message-passing supports on multi-core is currently a debated practice. The most common opinion is that it represents a poor choice for parallel programming ([54], [109], [185]), as the copies needed by message-passing communications can be unnecessary on such architectures.

One of the most important performance problems of the first implementation, is indeed that the worker, after computing the resulting task, *send* it to the collector. This means that we have a service time for each worker as follows:

$$T_{S-W_i} = T_{W-calc} + T_{copy}$$

Where T_{copy} represent a performance degradation w.r.t the sequential time. In our case, given the small grain of the computation, we expect T_{copy} to be an important fraction of T_{S-W_i}.

A very common way of masking communication latencies is to *delegate* the send operation to additional entities, so that we are able to *overlap* the computation (executed by the main process) and the communication (executed by this additional entity). Again, this is a well known concept in distributed-memory architectures, where the network interfaces are equipped of specific processors (communication units) able to execute the send/receive operation with almost no interaction with the main processor running the computation.

When communication inside shared-memory architectures is concerned, communication units can be concretized using the available architectural facilities:

- a communication unit can be implemented by a *core* of the architecture, dedicated to execute communication primitives delegated by different parts of the parallel computation (e.g. Emitter, Workers and Collector). In this case the core cannot be used to increase the parallelism degree of the application;
- instead of using an entire core, a communication unit can be a *hardware context* in the case of Hardware MultiThreaded CPUs;
- communication primitives can be delegated to specialized *co-processors* if they are available.

During our research we addressed the use of Hardware MultiThreading to overlap communications and computations[47] on the Intel Phi processor, with remarkable results: in most cases we were able to completely mask the communication overhead.

In this case, however, the Tile*Pro*64 does not offer hardware multithreading, nor specific co-processors, and we are not willing to use entire cores just to overlap communications. A different, yet interesting choice *specific* of the Emitter-Worker-Collector farm pattern, is to let the *Collector* execute the memory copy on the *receive* primitive. We therefore employ an asymmetric send/receive pattern in which:

- the **Emitter** copy the message during the send towards the workers.
- the **Workers** do not copy the message during the receive from the emitter, nor during the send towards the collector.
- the **Collector** copy the message during the receive from the workers.

This way, we expect the following service times:

$$\begin{aligned} T_{S-E} &= T_{copy} \\ T_{S-W_i} &= T_{W-calc} \\ T_{S-C} &= T_{copy} \end{aligned}$$

Where the worker now just execute the Sobel filter, while the cost of copying the data is only in charge of the supporting processes (i.e. Emitter and Collector).

It is important to notice that this approach maintain the semantic of local environment with message-passing: the *number* of memory copies remain the same, but we move the copy to a different entity of the parallel module.

Removing communications: Pointer Passing

The main problem of a farm implementation with a centralized emitter is that, in some cases, the emitter *may* become the bottleneck of the system, if $T_{copy} \geq {}^{T_{W-calc}}/_{n}$. This happen with an higher probability when the amount of workers - and thus of n - is significant (and with the Tile*Pro*64 we expect up to 46 workers). Decreasing the service time of the emitter helps in avoiding the problem; yet, in our cases, we cannot remove the T_{copy} as long as we work in a message passing environment. An interesting solution has been successfully used by the **FastFlow** framework[12], in which message passing is substituted with *pointer passing*. The basic idea is that, instead of copying the message from the sender to the receiver, the sender just pass the *pointer* to the message, and allow the receiver to directly work with the original copy.

Of course, this programming model make sense only under certain conditions. In particular, the sender **must** not use the message area once the pointer has been

forwarded, as that memory area will now be used *exclusively* by the receiver. In the case of the farm with an Emitter-Worker-Collector scheme this works quite well, as the Emitter and the Collector are just forwarding the messages, without actually modifying it. Other parallel patterns, however, may require a more complex approach to allow "pointer passing". An interesting example have been studied by our group for the parallelization of the Stream Join problem[46], where different workers share the same data, but a proper, cooperative update mechanism is required to ensure correctness. With this configuration we expect the following service times:

$$\begin{aligned} T_{S-E} &= \sim 0 \\ T_{S-W_i} &= T_{W-calc} \\ T_{S-C} &= \sim 0 \end{aligned}$$

Completely removing the possibility of making the Emitter (or the Collector) a bottleneck. This approach also have the benefit of removing the copies, thus lowering the pressure of the memory subsystem.

However, everything comes at a cost. In this case the messages are allocated *once* when sent to the emitter by the external module. In that moment, we have no way of knowing *which* worker will compute the task, so we cannot adopt the **NUMA-like RR** allocation policy. The best we can do is to let the sender spread the data across all the memory controllers, thus using the **SMP-like** policy. This will probably reduce the performance, especially when a small number of workers is used, because the memory latency is increased.

Fortunately, the use of pointer passing does not create big problems for disabling cache coherence, as at each moment of the execution only one entity is using a message.

To summarize, the absence of a local environment makes this implementation significantly different:

- We adopt the **SMP-like linear** allocation policy for the workers, as the NUMA-like RR is not applicable in this case.
- Automatic cache coherence can be still disabled, by inserting in specific points of the code cache control instructions.
- The Emitter and the Collector becomes very lightweight entities, as shown by the new implementation cost model, and thus will not become the bottleneck of the module.

8.3 Study of the message passing implementation

We start by summarizing the concepts already presented in Chapter 4) to evaluate the service time of this farm implementation. According to the performance model of the pattern, we have the following service times for each process:

$$\begin{aligned} T_{S-E} &= T_{copy} \\ T_{S-W_i} &= T_{W-calc} + T_{copy} \\ T_{S-C} &= \sim 0 \end{aligned}$$

In addition, because of the pipeling effect of the scheme we have that the global service time of the farm is given by

$$T_{S-FARM} = \max \left\{ T_{S-E}, \frac{T_{S-W_i}}{n}, T_{S-C} \right\} \tag{8.1}$$

Of course, to evaluate Eqn. 8.1 we need to estimate T_{W-calc} and T_{copy}.

Estimating T_{W-calc}

For all our test case we selected an image size of 3200×3200 pixels, resulting in an image of 10240000 Bytes.

To estimate the calculation time as

$$T_{w-calc} = Fixed\_Time + L_2\_misses \times Predicted\_Memory\_Latency$$

we need to extrapolate the two parameters $Fixed\_Time$ and L_2_misses from the original, sequential code. While the latter can be easily obtained by profiling the application, we do not have the same luck for $Fixed\_Time$, so we need apply a reverse approach: running the sequential application we obtain T^s_{w-calc} and $L_2\_misses^s$. The cache misses should be divided in *load miss* and *store miss* because, as described in Chapter 5, only load misses actually stall the processor. From these parameters we can derive

$$\begin{aligned} L_2\_misses &= L_2\_misses^s_{load} \\ Fixed\_Time &= T^s_{w-calc} - (L_2\_misses^s_{load} \times Measured\_Memory\_Latency) \end{aligned}$$

The execution of the sequential code, over a set of 100 images, resulted in the values reported in Table 8.1.

	Execution	Image	Estimated
T^s_{w-calc} (clocks)	24023521600	240235216	N/A
$L_2\_misses^s_{load}$	16318000	163180	160000
$L_2\_misses^s_{store}$	15995500	159955	160000

Table 8.1: Data from the execution of the sequential Sobel operator.

The first column contain the data directly gathered from the execution. From this, we calculated the statistics per image. The third column contains an estimation

of the misses considering the size of the image and of the cache lines, confirming the quality of the gathered statistics. Another good sign is the fact that the amount of *load miss* is approximately equal to the number of *store miss*: we read an image and save a new one on a different memory area, so we expected the number of misses on store and loads to be the same.

For $Measured\_Memory\_Latency$ we use the memory access latency of the architecture that, according to Chapter 5, should have an approximate value of 94 clock cycles. Using these data, we have that the parallel T_{w-calc} will be estimated as

$$\begin{aligned} L_2\_misses &= 163180 \\ Fixed\_Time &= 240235216 - (163180 \times 94) = 224896296 \\ T_{w-calc} &= 224896296 + (163180 \times Predicted\_Memory\_Latency) \end{aligned}$$

Estimating T_{copy}

For the memory copy execution time we could use the same approach; however, given the fact that this part of code is not written by the programmer, and can be considered an immutable part of the support, we decided to perform a different, more accurate study without the use of profiled execution. We created a specific copy implementation, that heavily exploit prefetching, loop unfolding and the VLIW instruction set to overlap the processing time with the memory transfers. The code is able to copy an entire L2 line in less than 80 clock cycles, and in the meantime prefetches the next input line. On top of this, thanks to the **wh64** instruction, we preallocate empty lines for the output, so that we do not generate misses on stores. We report the code that represents the loop over an L2 cache block in Listing 8.1.

Each line contain a bundle of instructions (up to 3) executed concurrently by the processor. Notice that we unfold the loop over the L1 cache line and we prefetch the data between L1 and L2. Of course, at the end of the copy we require to flush all the cache lines that belong to the output message, to guarantee that all the modified data reaches the memory. As introduced in Chapter 7, on the Tile*Pro*64 this can be done with specific assembler instructions (**flush**, **inv** and **finv** corresponding to the operations of flush, invalidate and both together) that work on single cache lines. However, given the relatively large size of the message w.r.t the L2 cache, it is more convenient to flush the whole L2 cache at the end of the message copy. We consider this part negligible for the evaluation of T_{copy}, as it is composed only of non-blocking write operations.

With this specific code, we are able to estimate T_{copy} as

$$T_{copy} = \frac{10240000}{64} \times Predicted\_Memory\_Latency$$

Because the computation is completely overlapped to memory transfers, so we have $Fixed\_Time = 0$. On the other hand, the number of memory transfers to read the message is exactly the size of the message divided by the L2 cache line size.

```
# Put input address in r7, put output address in r8
{ addi  r7,%2,0     addi  r8,%3,0                  }
# Calc. next input l2 line addr., alloc. output l2 line
{ addi  r5,r7,64    wh64  r8                       }
# i=16  , prefetch input l2
{ addi  r2,zero,16  lw    r5,r5                    }
# Calc. next input l1 line addr.
{ addi  r6,r7,16                                   }
# Loop over L1 lines
.START_INNER:
  # Prefetch the next L1 line
  { lw    r6,r6                                      }
  # load in, i = i - 4
  { lw  r3, r7      addi  r2,r2,-4                   }
  # store out, in = in+1
  { sw  r8,r3       addi  r7,r7,4                    }
  # load in,out = out+1
  { lw  r3, r7      addi  r8,r8,4                    }
  # store out  in = in+1
  { sw  r8,r3       addi  r7,r7,4                    }
  # load in   out = out+1
  { lw  r3, r7      addi  r8,r8,4                    }
  # store out,in = in+1
  { sw  r8,r3       addi  r7,r7,4                    }
  # load in,  out = out+1
  { lw  r3, r7      addi  r8,r8,4                    }
  # store out,in = in+1,r6=in+16
  { sw  r8,r3       addi  r7,r7,4   addi  r6,r7,20   }
  # out = out+1, jump if i>0
  { addi  r8,r8,4   bnzt  r2, .START_INNER           }
# End Inner Loop
# save new in address, save new out address
{addi %2,r7,0       addi  %3,r8,0                  }
```

Listing 8.1: Assembler implementation of message copy - loop over a L2 cache line.

8.3.1 Architecture Model Parameters

The last (but most important) step is to parameterize the architecture model of Chapter 5 for this application, in order to estimate *Predicted_Memory_Latency* (from now on R_q). The parameters of the model are summarized in Table 8.2. With these parameters we are able to simulate the queueing network model and obtain the values of R_q that will be used in evaluating T_{W-calc} and T_{copy}.

Given the pipelining of the entities, the module will be limited by the pipeline stage that represent the bottleneck. Different bottlenecks also means different parameters for our hardware model. For now on, we will *assume* that the Emitter and the Collector are not a bottleneck, and so the whole system will be limited by the throughput of the Workers, as we consider this the most probable case.

Parameter	Description	Multiplicity
n	Number of processors that generate memory requests	One
T_p	Computing time between two memory requests	One per core
p_{mod}	Probability of replacing a modified cache line	One per core
p_{mc1} p_{mc2} p_{mc3} p_{mc4}	Probability of sending a memory request towards the specified memory interface	One per core
k	Corrective factor to take into account the exact moment in which the processor enter and exit the stall (required to correctly estimate the memory latency)	One per core
p_W	Probability of receiving a memory write request	One per memory

Table 8.2: Summary of the parameters required to use the Tile*Pro*64 model.

For a specific parallelism degree m we will have, of course, m processors that behave as workers. In addition we have the Emitter and the Collector. In this implementation the Collector is negligible w.r.t the memory subsystem, so we can safely neglect it in the queueing network. In general, given the homogeneity of the workers, their parameters will be (mostly) the same, while the Emitter, performing significantly different, will also have different parameter values.

A different number of processors of the system also means different parameter values for the hardware model; thus we are studying a *family* of models, where the parameters are extracted in a common way but vary for each parallelism degree.

Placement on the mesh

To estimate the latency of each processor, it is important to correctly place the processors on the mesh, to use the right value for the various L_{net}.

Fortunately our Queueing Network model already contains the measured L_{net} parameters for each processor of the system, so we just need, for each core used, to enable the corresponding processor queue. In order to follow the **NUMA-like, RR** policy we will adopt the mapping scheme of Figure 8.5. Notice that we selected two *center* places for the Emitter and the Collector, so that their distance is practically the same w.r.t any memory interface.

Evaluating T_p

The selection of T_p is actually quite easy if we assume that the worker stage represent the bottleneck of the module.

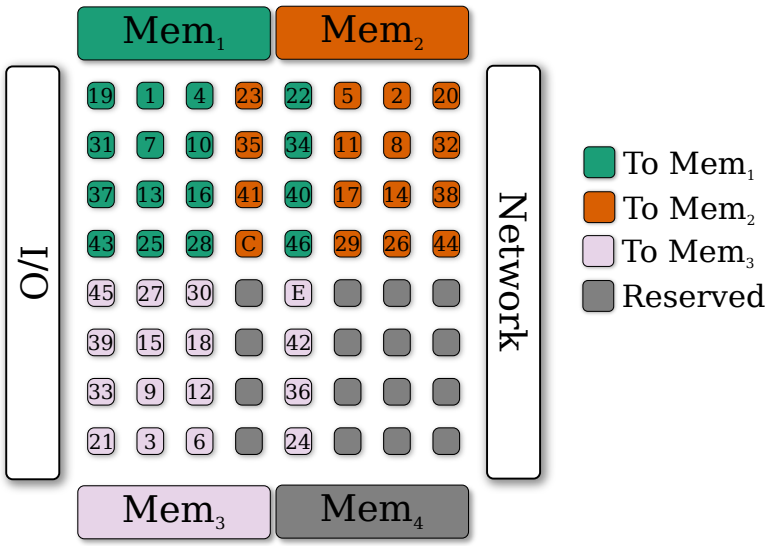


Figure 8.5: Allocation pattern for the processes in the message passing impl.

Workers Each worker will compute tasks continuously, because when a task is completed, it can immediately start with the next one assigned to him. We are certain of having assigned tasks because workers are the bottleneck of the module. Let's first consider the T_p for just T_{W-calc}; in this case we have that the processor works for $Fixed\_Time$. In this time it will generate $L_2\_misses^s_{load} + L_2\_misses^s_{store}$ read requests. Notice that now we include also the miss generated by store operations, as they do not concur in the estimation of T_{W-calc}, but are still required to correctly evaluate the load of the memory interfaces.

Using the previous parameters we have:

$$T_{p-W} = \frac{224896296}{(163180 + 159955)} = 695.98 \tag{8.2}$$

Now, for T_{copy}, we previously said that the calculation is completely overlapped with the memory requests; this means that, as soon as a response is received, a new request is sent out, resulting in a $T_p \sim 0$. So, in the overall T_{p-W}, we consider the same $Fixed\_Time$, but increase the amount of requests by adding those related to the memory copy. The memory copy just generate load misses because of the implementation so we have

$$T_{p-W} = \frac{224896296}{(163180 + 159955 + 160000)} = 465.49 \tag{8.3}$$

Emitter The Emitter, on the other hand, will just execute T_{copy} over and over again, really resulting in a $T_p \sim 0$. However, if we consider the fact that the workers are the bottleneck of the farm, we notice an interesting property: when the steady

state is reached, the emitter will *block*, after every send, waiting for one of the workers to complete their current task. When this happen the Emitter is free to send another message, then stops again and so on.

Let us make an example with a single worker. At steady state, the (single) worker is computing a task and the emitter is stopped. When the worker complete the task a space in the message channel is freed: the emitter execute a send and then stop again, waiting or another task to be completed. In fact, **the Emitter will send a message every** T_{S-W}. We can easily generalize the behavior by saying that in a parallel version the Emitter will send a message every T_{S-W}/n, that correspond to the service time of the whole farm: for each task computed, the Emitter will send a new message to a worker.

Now, we do not have the explicit T_{S-W} at this moment because we need the R_q of the model to estimate it. We believe that, at this point, $Fixed\_Time$ may be a good approximation. In this time we produce the memory read request of the copy routine, so we have

$$T_{p-E}^{(n)} = \frac{224896296}{n} \times \frac{1}{160000} = \frac{1405.60}{n} \tag{8.4}$$

Notice that the T_p of the Emitter is now parameterized with the number of workers, and tend to rapidly decrease: with $n = 10$ we have a $T_{p-E}^{(10)} = 140.56$, with $n = 20$ $T_{p-E}^{(20)} = 73.97$ and so on, ending with $T_{p-E}^{(46)} = 30.55$ clocks

Evaluating p_{mod}

Again, we study p_{mod} depending on the code to be executed. For T_{w-calc} we have that the program reads an image and store the result in another image (memory area). The two images are read linearly, so we expect that, at steady state, about half of the cache is occupied by the input image, and the other half by the output one. This means that, on a cache miss, the replaced block will have a probability of being in a modified state of $p_{mod} = 0.5$.

For T_{copy} the situation is slightly different: while at steady state the behavior seems very similar (we practically have an input and an output array again), we do not produce read requests for the stores and, because of the software cache coherence we will, at the end of the copy, flush the entire cache, producing new write requests. Approaching the problem from a different point of view we have that, during the copy, we will generate 160000 read requests for the input array and 160000 write requests for the output array. Put this in probabilities, we have $p_{mod} = 1$.

Workers The worker execute both the Sobel operator and the copy, so we need to average the two probabilities to model its behavior; we use a weighted average considering the amount of requests per operation ending with:

$$p_{mod-W} = \frac{((163180 + 159955) * 0.5) + (160000 * 1)}{163180 + 159955 + 160000} = 0.666 \tag{8.5}$$

Emitter On the other hand, the p_{mod} for the Emitter is very simple as it is composed only by the p_{mod} of the copy, so we have

$$p_{mod-E} = 1 \tag{8.6}$$

Evaluating p_{mc}

To estimate p_{mc} we first need to decide where we will allocate the buffers to hold the messages and the local copies of the workers. Given the choice of using the **NUMA-like** allocation policy, each buffer should be allocated near to the core that uses it. So, the local buffer of the worker (containing the filtered image) and the channel buffer for the Emitter-Worker communication (containing the input images) will be allocated in the "local memory" of the worker. We decided to do the same for the channel buffer for the Worker-Collector communication, so that we reduce as much as possible T_{S-W_i}.

For the Emitter, we already set the channels towards the Workers on their memories. We still need to decide for the allocation of the *input channel* of the Emitter. Given its equal distance to the various memory controllers, we decided to use the third.

Workers Given the selected allocation, each worker will access only to its "local" memory, so we have $p_{mcx} = 1$ for the single memory interface used by the specific worker. In particular, given the round-robin process allocation, we have (using the notation p_{mcx-W_y} to indicate the probability of controller x for the worker y):

$$\begin{aligned}
p_{mc1-W_1} = 1, p_{mc2-W_1} = 0, p_{mc3-W_1} = 0, p_{mc4-W_1} = 0 \\
p_{mc1-W_2} = 0, p_{mc2-W_2} = 1, p_{mc3-W_2} = 0, p_{mc4-W_2} = 0 \\
p_{mc1-W_3} = 0, p_{mc2-W_3} = 0, p_{mc3-W_3} = 1, p_{mc4-W_3} = 0 \\
p_{mc1-W_4} = 1, p_{mc2-W_4} = 0, p_{mc3-W_4} = 0, p_{mc4-W_4} = 0
\end{aligned}$$

and so on for each worker. Clearly p_{mc4} is always 0 because we decided to reserve it for other parts of the application

Emitter The pattern for the Emitter is a bit more complicate, as the input data will always reside on $mc3$, while the output data will be sent to the three controllers, depending on the number of workers.

In particular, with a single worker, we have that the output data will reside only on $mc1$, and the input data on $mc3$, thus (using the notation $p_{mcx-E^{(y)}}$ to indicate the probability of controller x for the Emitter with y workers):

$$p_{mc1-E^{(1)}} = 0.5, p_{mc2-E^{(1)}} = 0, p_{mc3-E^{(1)}} = 0.5, p_{mc4-E^{(1)}} = 0$$

In general, half of the accesses will be directed to $mc3$ for reading the input array; the other half will be distributed to the three interfaces, depending on how many workers

uses that interface. For example, with two workers, we have one output array in $mc1$ and the other in $mc2$; with three workers one output array per interface. With four workers, two of them will use $mc1$, one $mc2$, and one $mc3$, ending with:

$$p_{mc1-E^{(2)}} = 0.25, p_{mc2-E^{(2)}} = 0.25, p_{mc3-E^{(2)}} = 0.5, p_{mc4-E^{(2)}} = 0$$
$$p_{mc1-E^{(3)}} = 0.16, p_{mc2-E^{(3)}} = 0.16, p_{mc3-E^{(3)}} = 0.66, p_{mc4-E^{(3)}} = 0$$
$$p_{mc1-E^{(4)}} = 0.25, p_{mc2-E^{(4)}} = 0.125, p_{mc3-E^{(4)}} = 0.625, p_{mc4-E^{(4)}} = 0$$

And so on. With this pattern we can easily notice an asymmetry of the requests, that could unbalance of the memory system, ending with $mc3$ more loaded than the others. In fact, a more wisely decision could have been to uniformly distribute the input data of the Emitter on the three interfaces, using a mixed SMP-like and NUMA-like allocation, to balance better the memory load.

Evaluating k

A proper value of k is required to correctly estimate R_q. To evaluate k we should need a proper methodology by analyzing the source code; unfortunately, at the moment this is missing, so to obtain k we just run the parallel code with a single worker and calculate k as the difference between the modeled system with $k = 0$ and the measured R_q. In particular we have, independently of the parallelism degree, the following values:

$$k_E = 18$$
$$k_{Wi} = 23$$

Evaluating p_W

Lastly, we need to evaluate the p_W values for each memory controller, depending on the percent of read and write requests received.

We start the reasoning with a single worker, and then try to generalize it. With a single worker we have that, for each image, the worker will generate $163180 + 159955 + 160000$ requests for the calculation, plus $2 * 160000$ requests for the copy, all directed to the first memory controller. Of these, $2 * 160000$ correspond to write operations. On the very same controller, the Emitter will produce the requests for filling the message buffer, that correspond to other 160000 write requests. On the contrary, on the third controller, we have only the read requests produced by the Emitter:

$$p_{W1-1} = \frac{3 * 160000}{3 * 160000 + 163180 + 159955 + 160000} = 0.499, p_{W3-1} = 0$$

With two workers the situation is quite similar: the requests for the first memory controller are the same as before (produced by the first worker and the emitter), the second memory controller host a different worker, so we apply the previous

approach (and we end with the very same result), while the third is still having only read requests.

$$p_{W1-2} = 0.499, p_{W2-2} = 0.499, p_{W3-2} = 0, p_{W4-2} = 0$$

With three workers things become a little more complex: the first two controllers maintain the same values, but for the third we now have a worker *and* the emitter read requests. For each image we produce, as before, $3 * 160000 + 163180 + 159955 + 160000$ requests, of which $3 * 160000$ write ops. During an image filtering, however, the farm produces three results (its service time), so emitter will also read three messages, producing other $3 * 160000$ read requests. Thus

$$p_{W1-2} = 0.499, p_{W2-2} = 0.499, p_{W3-2} = \frac{3 * 160000}{7 * 160000 + 163180 + 159955} = 0.33$$

A different, more general approach can be obtained by using T_p and p_{mod} for each processor to evaluate the amount of read and write requests generate by each core per clock. Multiplying them by p_{mcx} we obtain the requests per memory controllers and finally, summing the data of all the processors, calculate the global p_{W1}, p_{W2} and p_{W3}, for each parallelism degree.

Parameter Summary

For convenience, here we report all the parameters in a concise form: Table 8.3 collect the values dependent on the number of workers while the other, parallelism-independent parameters are:

$$\begin{aligned} T_{p-W} &= 465.49 \\ p_{mod-W} &= 0.666 \\ k_E &= 18 \\ k_W &= 23 \end{aligned}$$

Table 8.3: Model parameters for the Message Passing Implementation, per parallelism degree.

Par.	Emitter				Memory Interfaces			Par.	Emitter				Memory Interfaces		
Degree	T_p	p_{mc1}	p_{mc2}	p_{mc3}	p_{W1}	p_{W2}	p_{W3}	Degree	T_p	p_{mc1}	p_{mc2}	p_{mc3}	p_{W1}	p_{W2}	p_{W3}
1	1405,60	0,50	0	0,50	0,50	0	0	24	58,57	0,17	0,17	0,67	0,50	0,50	0,33
2	702,80	0,25	0,25	0,50	0,50	0,50	0	25	56,22	0,18	0,16	0,66	0,50	0,50	0,33
3	468,53	0,17	0,17	0,67	0,50	0,50	0,33	26	54,06	0,17	0,17	0,65	0,50	0,50	0,32
4	351,40	0,25	0,13	0,63	0,50	0,50	0,30	27	52,06	0,17	0,17	0,67	0,50	0,50	0,33
5	281,12	0,20	0,20	0,60	0,50	0,50	0,27	28	50,20	0,18	0,16	0,66	0,50	0,50	0,33
6	234,27	0,17	0,17	0,67	0,50	0,50	0,33	29	48,47	0,17	0,17	0,66	0,50	0,50	0,33
7	200,80	0,21	0,14	0,64	0,50	0,50	0,32	30	46,85	0,17	0,17	0,67	0,50	0,50	0,33
8	175,70	0,19	0,19	0,63	0,50	0,50	0,30	31	45,34	0,18	0,16	0,66	0,50	0,50	0,33
9	156,18	0,17	0,17	0,67	0,50	0,50	0,33	32	43,93	0,17	0,17	0,66	0,50	0,50	0,33
10	140,56	0,20	0,15	0,65	0,50	0,50	0,32	33	42,59	0,17	0,17	0,67	0,50	0,50	0,33
11	127,78	0,18	0,18	0,64	0,50	0,50	0,31	34	41,34	0,18	0,16	0,66	0,50	0,50	0,33
12	117,13	0,17	0,17	0,67	0,50	0,50	0,33	35	40,16	0,17	0,17	0,66	0,50	0,50	0,33
13	108,12	0,19	0,15	0,65	0,50	0,50	0,32	36	39,04	0,17	0,17	0,67	0,50	0,50	0,33
14	100,40	0,18	0,18	0,64	0,50	0,50	0,32	37	37,99	0,18	0,16	0,66	0,50	0,50	0,33
15	93,71	0,17	0,17	0,67	0,50	0,50	0,33	38	36,99	0,17	0,17	0,66	0,50	0,50	0,33
16	87,85	0,19	0,16	0,66	0,50	0,50	0,33	39	36,04	0,17	0,17	0,67	0,50	0,50	0,33
17	82,68	0,18	0,18	0,65	0,50	0,50	0,32	40	35,14	0,18	0,16	0,66	0,50	0,50	0,33
18	78,09	0,17	0,17	0,67	0,50	0,50	0,33	41	34,28	0,17	0,17	0,66	0,50	0,50	0,33
19	73,98	0,18	0,16	0,66	0,50	0,50	0,33	42	33,47	0,17	0,17	0,67	0,50	0,50	0,33
20	70,28	0,18	0,18	0,65	0,50	0,50	0,32	43	32,69	0,17	0,16	0,66	0,50	0,50	0,33
21	66,93	0,17	0,17	0,67	0,50	0,50	0,33	44	31,95	0,17	0,17	0,66	0,50	0,50	0,33
22	63,89	0,18	0,16	0,66	0,50	0,50	0,33	45	31,24	0,17	0,17	0,67	0,50	0,50	0,33
23	61,11	0,17	0,17	0,65	0,50	0,50	0,32	46	30,56	0,17	0,16	0,66	0,50	0,50	0,33

8.3.2 Predicted Service Times

Now we can solve the queueing network system by using **EQNSim** and obtain the R_q values of interest. In particular, we decided to evaluate the specific R_q of the Emitter and the average R_q of all the Workers. These two values, reported in Table 8.4, are used to obtain the predicted values of T_{S-E}, T_{S-W_i} and T_S. These are reported in Table 8.5, with the comparison w.r.t. the times obtained by the real execution.

Par.	R_q		Par.	R_q	
Degree	Worker	Emitter	Degree	Worker	Emitter
1	115,00	123,92	24	149,87	162,00
2	115,66	123,97	25	152,18	163,59
3	116,21	124,21	26	154,04	164,53
4	116,86	125,71	27	157,69	169,73
5	117,27	127,31	28	160,17	172,27
6	117,55	128,71	29	162,13	171,68
7	120,01	130,01	30	166,17	179,72
8	120,25	130,19	31	168,83	181,86
9	122,20	132,44	32	171,69	183,75
10	123,76	133,40	33	176,06	193,48
11	125,15	133,41	34	179,79	194,88
12	126,92	136,97	35	182,51	194,59
13	128,51	137,83	36	187,21	202,75
14	128,98	139,17	37	190,58	204,33
15	131,55	141,93	38	194,57	210,43
16	133,37	143,28	39	199,73	215,75
17	134,86	144,18	40	204,25	219,19
18	137,28	148,15	41	207,70	221,03
19	139,19	149,33	42	214,73	234,21
20	140,86	149,97	43	219,45	237,81
21	143,18	153,45	44	224,46	239,42
22	145,23	155,78	45	230,47	251,32
23	147,02	157,18	46	236,30	255,14

Table 8.4: R_q values obtained by solving the Queueing Network model parameterized for the message passing implementation. Times in clock cycles.

Table 8.5: Model values of T_{S-E}, T_{S-W_i} and T_S for the message passing imp., w.r.t the execution. Times in clock cycles.

Par. Degree	T_{S-E} Expected	T_{S-E} Measured	T_{S-E} % Error	T_{S-W_i} Expected	T_{S-W_i} Measured	T_{S-W_i} % Error	T_S Expected	T_S Measured	T_S % Error
1	19,83M	19,85M	0,12	262,06M	262,10M	0,01	262,06M	262,34M	0,11
2	19,84M	19,82M	0,08	262,27M	262,17M	0,04	131,14M	131,27M	0,10
3	19,87M	20,81M	4,51	262,45M	262,53M	0,03	87,48M	88,03M	0,62
4	20,11M	20,85M	3,54	262,66M	263,08M	0,16	65,67M	66,05M	0,59
5	20,37M	20,70M	1,59	262,80M	263,33M	0,20	52,56M	52,88M	0,60
6	20,59M	21,39M	3,73	262,88M	263,77M	0,34	43,81M	44,33M	1,16
7	20,80M	21,43M	2,92	263,68M	264,52M	0,32	37,67M	38,06M	1,02
8	20,83M	21,31M	2,24	263,76M	265,14M	0,52	32,97M	33,37M	1,21
9	21,19M	21,79M	2,73	264,39M	265,75M	0,51	29,38M	29,79M	1,39
10	21,34M	21,75M	1,88	264,89M	266,34M	0,54	26,49M	26,88M	1,45
11	21,35M	21,67M	1,49	265,34M	266,87M	0,57	24,12M	24,50M	1,54
12	21,92M	22,21M	1,33	265,91M	267,83M	0,71	22,16M	22,59M	1,89
13	22,05M	22,13M	0,34	266,43M	267,61M	0,44	22,05M	22,36M	1,38
14	22,27M	21,98M	1,31	266,58M	267,71M	0,42	22,27M	22,20M	0,31
15	22,71M	22,21M	2,26	267,41M	267,97M	0,21	22,71M	22,41M	1,32
16	22,93M	22,18M	3,35	268,00M	268,03M	0,01	22,93M	22,38M	2,45
17	23,07M	22,03M	4,71	268,48M	267,95M	0,20	23,07M	22,21M	3,84
18	23,70M	22,30M	6,28	269,26M	268,12M	0,43	23,70M	22,48M	5,46
19	23,89M	22,23M	7,48	269,88M	268,01M	0,70	23,89M	22,40M	6,65
20	24,00M	22,17M	8,24	270,42M	268,09M	0,87	24,00M	22,33M	7,45
25	26,17M	22,37M	17,01	274,08M	268,37M	2,13	26,17M	22,51M	16,26
30	28,76M	22,41M	28,31	278,60M	268,41M	3,80	28,76M	22,54M	27,59
35	31,13M	22,33M	39,41	283,88M	268,33M	5,79	31,13M	22,45M	38,69
40	35,07M	22,29M	57,33	290,91M	268,12M	8,50	35,07M	22,40M	56,59
46	40,82M	22,39M	82,37	301,27M	268,51M	12,20	40,82M	22,49M	81,52

The results are indeed excellent in the first values, up to about $13 - 14$ workers where the error on T_S is always lower than $\sim 2\%$; however, when the number of processing cores is higher, the error start to increase, reaching a (virtually) unacceptable error of $\sim 81\%$. This behavior, however, was expected: with more than 12 workers, the bottleneck moves from the worker to the emitter stage. In this new scenario the parameters selected for our hardware model are no longer valid, as **we made the assumption that the emitter was not the bottleneck**; this produces wrong predictions of R_q that affect the cost model.

In the real execution the service time of the emitter is not further increased when it becomes the bottleneck: intuitively the R_q of the system is not growing anymore by adding more workers because they are slowed down by the emitter: we have more workers, but with an increased T_p. To handle the emitter bottleneck problem we could, of course, re-parameterize the system from scratch; however, a faster and still quite accurate solution consist in cutting the service time values to the first one in which the Emitter is the bottleneck. In this point the estimated R_q is quite good (as we are not having an excess of requests because the Emitter has just become the limiting stage of the pipeline), and thus we expect a good approximation for the T_S. For any parallelism degree greater than this point, we always use the threshold T_S value. This way we obtain the model predictions of Table 8.6 that improve the overall prediction, ending with an error always lower than $\sim 2.3\%$. We thus consider this model accurate enough for our purposes. A graphical comparison of the modeled and measured T_S is also showed in Figure 8.6.

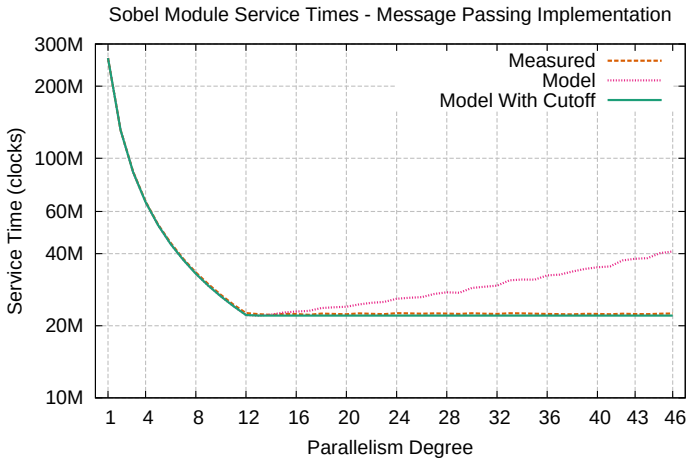


Figure 8.6: Comparison between architecture and model values of T_S for the message passing implementation of the Sobel Module. Times in clock cycles.

Table 8.6: Comparison between architecture and final model values of T_S for the message passing implementation of the Sobel Module. Times in clock cycles.

Par.	T_S		%	Par.	T_S		%
Degree	Expected	Measured	Error	Degree	Expected	Measured	Error
1	262,06M	262,34M	0,11	24	22,05M	22,57M	2,29
2	131,14M	131,27M	0,10	25	22,05M	22,51M	2,05
3	87,48M	88,03M	0,62	26	22,05M	22,45M	1,76
4	65,67M	66,05M	0,59	27	22,05M	22,51M	2,05
5	52,56M	52,88M	0,60	28	22,05M	22,47M	1,86
6	43,81M	44,33M	1,16	29	22,05M	22,39M	1,52
7	37,67M	38,06M	1,02	30	22,05M	22,54M	2,16
8	32,97M	33,37M	1,21	31	22,05M	22,44M	1,71
9	29,38M	29,79M	1,39	32	22,05M	22,40M	1,54
10	26,49M	26,88M	1,45	33	22,05M	22,53M	2,13
11	24,12M	24,50M	1,54	34	22,05M	22,51M	2,04
12	22,16M	22,59M	1,89	35	22,05M	22,45M	1,77
13	22,05M	22,36M	1,38	36	22,05M	22,40M	1,56
14	22,05M	22,20M	0,66	37	22,05M	22,36M	1,39
15	22,05M	22,41M	1,61	38	22,05M	22,31M	1,17
16	22,05M	22,38M	1,45	39	22,05M	22,41M	1,60
17	22,05M	22,21M	0,73	40	22,05M	22,40M	1,53
18	22,05M	22,48M	1,89	41	22,05M	22,34M	1,27
19	22,05M	22,40M	1,57	42	22,05M	22,45M	1,77
20	22,05M	22,33M	1,25	43	22,05M	22,38M	1,45
21	22,05M	22,51M	2,03	44	22,05M	22,34M	1,30
22	22,05M	22,41M	1,60	45	22,05M	22,44M	1,73
23	22,05M	22,36M	1,39	46	22,05M	22,49M	1,94

8.4 Study of the message passing implementation with copy on receive

With this implementation we maintain most of the characteristics of the previous one, but try to reduce the service time of the workers by delegating the message copy to the receiver, so that we are able to overlap T_{W-calc} and T_{copy}. According to the performance model of the pattern, we now have the following service times for each process:

$$\begin{aligned} T_{S-E} &= T_{copy} \\ T_{S-W_i} &= T_{W-calc} \\ T_{S-C} &= T_{copy} \end{aligned}$$

In addition, because of the pipeling effect of the scheme we have that the global service time of the farm is still given by

$$T_{S-FARM} = \max\left\{T_{S-E}, \frac{T_{S-W_i}}{n}, T_{S-C}\right\}$$

The previous estimations of T_{W-calc} and T_{copy} still holds, because the sequential parts of the code are exactly the same:

$$\begin{aligned} T_{w-calc} &= 224896296 + (163180 \times Predicted\_Memory\_Latency) \\ T_{copy} &= \frac{10240000}{64} \times Predicted\_Memory\_Latency \end{aligned}$$

8.4.1 Architecture Model Parameters

The change in the implementation require us to verify all the model parameters, and eventually change them to match the new behavior. We will therefore review each points.

We still *assume* that the Emitter and the Collector are not a bottleneck, so that the whole system will be limited by the throughput of the Workers. Now we know that this represent a good approach for the whole model, as it allow us to find the point in which the emitter or the collector becomes the bottleneck; from that point we can approximate the service time of the farm by using the cutoff method previously described.

In this version for a specific parallelism degree m we will have m processors that behave as workers, plus the Emitter and the Collector. In this implementation we cannot consider the Collector negligible, as it plays an active role in the computation; this means that, w.r.t. the previous case, we enable an additional processor in the model and, of course, we will need to evaluate its parameters.

Placement on the mesh

With this implementation we are just interested in overlapping the communications, so we keep exactly the same process allocation policy, where the Emitter and the Collector are kept at the center of the mesh.

Evaluating T_p

In the selection of T_p important differences start to emerge w.r.t the previous implementation.

Workers In this case the workers just compute the tasks, without paying the cost of copying the message. Thus, w.r.t the previous implementation, we do not consider the memory requests to copy the message and we have

$$T_{p-W} = \frac{224896296}{(163180 + 159955)} = 695.98$$

Emitter The Emitter, on the other hand, will just execute T_{copy} over and over again as in the previous implementation; we apply the same considerations about the behavior with the bottleneck on the workers and we end with the same values as before:

$$T_{p-E}^{(n)} = \frac{224896296}{n} \times \frac{1}{160000} = \frac{1405.60}{n}$$

Collector With this implementation we also need to evaluate the T_p of the Collector. Here we can apply the same approach of the Emitter: in theory, a Collector is able to copy messages continuously, resulting in a $T_{p-C} \simeq 0$. However, as the Emitter, its throughput is limited by the one of the Workers and, as the Emitter, will perform a copy every T_{S-W}. We can use the very same approximation used for the Emitter and have

$$T_{p-C}^{(n)} = \frac{224896296}{n} \times \frac{1}{160000} = T_{p-E}^{(n)} = \frac{1405.60}{n}$$

Evaluating p_{mod}

The values for p_{mod} of T_{w-calc} and T_{copy} are the same (because inherited from the sequential code):

$$p_{mod}^{calc} = 0.5$$
$$p_{mod}^{copy} = 1$$

Workers With this implementation the worker just execute the Sobel operator, so we do not need to average the probabilities and just have

$$p_{mod-W} = 0.5$$

Emitter The p_{mod} for the Emitter still the same, as its behavior has not changed:

$$p_{mod-E} = 1$$

Collector The p_{mod} for the Collector can be evaluated by considering that, as the Emitter, it just execute the copy of messages, so

$$p_{mod-C} = 1$$

Evaluating p_{mc}

For the estimation of p_{mc} we consider the fact that the memory areas are allocated *exactly* as in the previous case

Workers Again, each worker will access only to its "local" memory, so we have $p_{mcx} = 1$ for the single memory interface used by the specific worker and, given the round-robin process allocation, we have:

$$\begin{aligned}
p_{mc1-W_1} = 1, p_{mc2-W_1} = 0, p_{mc3-W_1} = 0, p_{mc4-W_1} = 0 \\
p_{mc1-W_2} = 0, p_{mc2-W_2} = 1, p_{mc3-W_2} = 0, p_{mc4-W_2} = 0 \\
p_{mc1-W_3} = 0, p_{mc2-W_3} = 0, p_{mc3-W_3} = 1, p_{mc4-W_3} = 0 \\
p_{mc1-W_4} = 1, p_{mc2-W_4} = 0, p_{mc3-W_4} = 0, p_{mc4-W_4} = 0
\end{aligned}$$

and so on for each worker.

Emitter The pattern for the Emitter still the same: read the input data on mc_3, store the output data on the three controllers, depending on the number of workers:

$$\begin{aligned}
p_{mc1-E^{(1)}} = 0.5, p_{mc2-E^{(1)}} = 0, p_{mc3-E^{(1)}} = 0.5, p_{mc4-E^{(1)}} = 0 \\
p_{mc1-E^{(2)}} = 0.25, p_{mc2-E^{(2)}} = 0.25, p_{mc3-E^{(2)}} = 0.5, p_{mc4-E^{(2)}} = 0 \\
p_{mc1-E^{(3)}} = 0.16, p_{mc2-E^{(3)}} = 0.16, p_{mc3-E^{(3)}} = 0.66, p_{mc4-E^{(3)}} = 0 \\
p_{mc1-E^{(4)}} = 0.25, p_{mc2-E^{(4)}} = 0.125, p_{mc3-E^{(4)}} = 0.625, p_{mc4-E^{(4)}} = 0
\end{aligned}$$

and so on.

Collector The pattern of the Collector is new and needs to be studied. If we keep the very same memory allocation, we have that, for each message to be copied, both the input and the output buffer are allocated near to the worker that logically sent the message. So, with one worker we have:

$$p_{mc1-C^{(1)}} = 1, p_{mc2-C^{(1)}} = 0, p_{mc3-C^{(1)}} = 0, p_{mc4-C^{(1)}} = 0$$

because the only worker is allocated next to the first memory controller. With two workers we have to handle half of the messages allocated on $mc1$, the other half on $mc2$; we generalize the reasoning and have

$$p_{mc1-C^{(2)}} = 0.5, p_{mc2-C^{(2)}} = 0.5, p_{mc3-C^{(2)}} = 0, p_{mc4-C^{(2)}} = 0$$
$$p_{mc1-C^{(3)}} = 0.33, p_{mc2-C^{(3)}} = 0.33, p_{mc3-C^{(3)}} = 0.33, p_{mc4-C^{(3)}} = 0$$
$$p_{mc1-C^{(4)}} = 0.5, p_{mc2-C^{(4)}} = 0.25, p_{mc3-C^{(4)}} = 0.25, p_{mc4-C^{(4)}} = 0$$

It should be noted that, w.r.t. the Emitter, the Collector does balance better the requests and, when using the same number of worker per controller, is able to uniformly distribute them.

Evaluating k

The values for k, still obtained by executing the parallel implementation with parallelism degree of one, are:

$$k_E = 18$$
$$k_C = 27$$
$$k_{Wi} = 3$$

Evaluating p_W

Lastly, we need to evaluate the p_W values for each memory controller, depending on the percent of read and write requests received. W.r.t the simple message passing implementation we just moved the copy operation, but kept the same memory locations and access pattern. Therefore we expect the very same number of read and write requests and thus the same values of p_W of the previous implementation.

Parameter Summary

For convenience, here we report all the parameters in a concise form: Table 8.7 collect the values dependent on the number of workers[1] while the other, parallelism-independent parameters are

$$\begin{aligned} T_{p-W} &= 695.98 \\ p_{mod-W} &= 0.5 \\ k_E &= 18 \\ k_C &= 27 \\ k_{Wi} &= 3 \end{aligned}$$

[1]We omitted the p_{wx} values as they are exactly the same of the previous implementation

Table 8.7: Model parameters for the Copy on Receive Message Passing Implementation, per parallelism degree.

Par. Degree	T_{p-E} T_{p-C}	Emitter			Collector			Par. Degree	T_{p-E} T_{p-C}	Emitter			Collector		
		p_{mc1}	p_{mc2}	p_{mc3}	p_{mc1}	p_{mc2}	p_{mc3}			p_{mc1}	p_{mc2}	p_{mc3}	p_{mc1}	p_{mc2}	p_{mc3}
1	1405.60	0.5	0	0.5	1	0	0	24	58.57	0.17	0.17	0.67	0.33	0.33	0.33
2	702.80	0.25	0.25	0.5	0.5	0.5	0	25	56.22	0.18	0.16	0.66	0.36	0.32	0.32
3	468.53	0.17	0.17	0.67	0.33	0.33	0.33	26	54.06	0.17	0.17	0.65	0.35	0.35	0.31
4	351.40	0.25	0.13	0.63	0.50	0.25	0.25	27	52.06	0.17	0.17	0.67	0.33	0.33	0.33
5	281.12	0.20	0.20	0.60	0.40	0.40	0.20	28	50.20	0.18	0.16	0.66	0.36	0.32	0.32
6	234.27	0.17	0.17	0.67	0.33	0.33	0.33	29	48.47	0.17	0.17	0.66	0.34	0.34	0.31
7	200.80	0.21	0.14	0.64	0.43	0.29	0.29	30	46.85	0.17	0.17	0.67	0.33	0.33	0.33
8	175.70	0.19	0.19	0.63	0.38	0.38	0.25	31	45.34	0.18	0.16	0.66	0.35	0.32	0.32
9	156.18	0.17	0.17	0.67	0.33	0.33	0.33	32	43.93	0.17	0.17	0.66	0.34	0.34	0.31
10	140.56	0.20	0.15	0.65	0.40	0.30	0.30	33	42.59	0.17	0.17	0.67	0.33	0.33	0.33
11	127.78	0.18	0.18	0.64	0.36	0.36	0.27	34	41.34	0.18	0.16	0.66	0.35	0.32	0.32
12	117.13	0.17	0.17	0.67	0.33	0.33	0.33	35	40.16	0.17	0.17	0.66	0.34	0.34	0.31
13	108.12	0.19	0.15	0.65	0.38	0.31	0.31	36	39.04	0.17	0.17	0.67	0.33	0.33	0.33
14	100.40	0.18	0.18	0.64	0.36	0.36	0.29	37	37.99	0.18	0.16	0.66	0.35	0.32	0.32
15	93.71	0.17	0.17	0.67	0.33	0.33	0.33	38	36.99	0.17	0.17	0.66	0.34	0.34	0.32
16	87.85	0.19	0.16	0.66	0.38	0.31	0.31	39	36.04	0.17	0.17	0.67	0.33	0.33	0.33
17	82.68	0.18	0.18	0.65	0.35	0.35	0.29	40	35.14	0.18	0.16	0.66	0.35	0.33	0.33
18	78.09	0.17	0.17	0.67	0.33	0.33	0.33	41	34.28	0.17	0.17	0.66	0.34	0.34	0.32
19	73.98	0.18	0.16	0.66	0.37	0.32	0.32	42	33.47	0.17	0.17	0.67	0.33	0.33	0.33
20	70.28	0.18	0.18	0.65	0.35	0.35	0.30	43	32.69	0.17	0.16	0.66	0.35	0.33	0.33
21	66.93	0.17	0.17	0.67	0.33	0.33	0.33	44	31.95	0.17	0.17	0.66	0.34	0.34	0.32
22	63.89	0.18	0.16	0.66	0.36	0.32	0.32	45	31.24	0.17	0.17	0.67	0.33	0.33	0.33
23	61.11	0.17	0.17	0.65	0.35	0.35	0.30	46	30.56	0.17	0.16	0.66	0.35	0.33	0.33

8.4.2 Predicted Service Times

Now we can solve the queueing network system by using **EQNSim** and obtain the R_q values of interest. In particular, we decided to evaluate the specific R_q of the Emitter and the Collector, and the average R_q of all the Workers. These values, reported in Table 8.8, are used to obtain the prediction of T_{S-E}, T_{S-C}, T_{S-W_i} and T_S in Table 8.9.

Par.		R_q		Par.		R_q	
Degree	Worker	Emitter	Collector	Degree	Worker	Emitter	Collector
1	96	124.385	130	24	122.851	150.372	161.043
2	96.9063	124.134	130.821	25	124.359	150.607	162.1
3	97.4375	124.52	134.982	26	125.774	151.885	162.899
4	99.9455	125.758	135.369	27	127.732	155.421	173.791
5	99.8705	126.481	134.527	28	128.982	155.75	170.483
6	99.5967	127.387	137.247	29	130.088	156.18	167.764
7	101.798	128.415	138.974	30	132.439	159.68	172.028
8	102.165	127.649	139.712	31	134.094	162.452	171.903
9	102.734	130.785	141.21	32	135.14	161.612	173.296
10	105.117	131.488	142.551	33	137.73	165.48	176.797
11	105.614	132.069	142.999	34	139.464	167.887	176.683
12	106.833	134.002	144.501	35	140.69	167.71	178.008
13	107.489	134.739	145.247	36	143.415	172.111	181.087
14	109.589	135.384	146.628	37	145.237	172.613	183.647
15	111.067	137.878	148.457	38	146.736	174.39	184.153
16	112.125	138.424	149.348	39	149.969	181.776	188.02
17	113.056	138.991	150.255	40	151.729	180.485	190.886
18	114.491	141.953	152.123	41	153.293	182.824	189.161
19	115.726	142.623	153.283	42	156.308	187.659	192.734
20	116.932	144.812	154.468	43	158.931	184.493	194.991
21	119.219	146.435	155.835	44	160.329	187.635	196.773
22	119.831	146.602	157.66	45	164.389	193.504	202.239
23	121.226	147.089	159.215	46	166.221	195.61	199.166

Table 8.8: R_q values obtained by solving the Queueing Network model parameterized for the copy on receive message passing implementation. Times in clock cycles.

Table 8.9: Model values of T_{S-E}, T_{S-W_i} and T_S for the copy on recv. mp imp., w.r.t the execution. Times in clock cycles.

Par.	T_{S-E}			T_{S-C}			T_{S-W_i}			T_S		
Degree	Exp.	Meas.	% E.	Exp.	Meas.	% E.	Exp.	Meas.	% E.r	Exp.	Meas.	% E.
1	19.90M	19.86M	0.21	20.80M	20.76M	0.19	240.56M	240.40M	0.07	240.56M	240.87M	0.13
2	19.86M	19.83M	0.16	20.93M	20.83M	0.50	240.71M	240.38M	0.14	120.35M	120.56M	0.17
3	19.92M	21.13M	5.71	21.60M	21.51M	0.40	240.80M	241.26M	0.19	80.27M	81.09M	1.01
4	20.12M	20.87M	3.59	21.66M	21.52M	0.65	241.21M	241.76M	0.23	60.30M	60.87M	0.94
5	20.24M	20.85M	2.93	21.52M	21.91M	1.77	241.19M	242.15M	0.40	48.24M	48.81M	1.18
6	20.38M	21.37M	4.63	21.96M	22.25M	1.29	241.15M	242.68M	0.63	40.19M	40.83M	1.57
7	20.55M	21.25M	3.29	22.24M	22.43M	0.88	241.51M	243.38M	0.77	34.50M	35.13M	1.78
8	20.42M	21.13M	3.32	22.35M	22.55M	0.88	241.57M	243.78M	0.91	30.20M	30.81M	1.98
9	20.93M	21.69M	3.52	22.59M	22.81M	0.97	241.66M	244.54M	1.18	26.85M	27.52M	2.45
10	21.04M	21.62M	2.71	22.81M	23.01M	0.86	242.05M	245.18M	1.28	24.20M	24.84M	2.57
11	21.13M	21.65M	2.42	22.88M	23.14M	1.13	242.13M	245.58M	1.40	22.88M	23.42M	2.31
12	21.44M	22.11M	3.01	23.12M	23.31M	0.82	242.33M	245.76M	1.40	23.12M	23.61M	2.06
13	21.56M	21.86M	1.39	23.24M	23.43M	0.80	242.44M	245.89M	1.41	23.24M	23.69M	1.91
14	21.66M	21.79M	0.59	23.46M	23.36M	0.44	242.78M	245.83M	1.24	23.46M	23.61M	0.64
15	22.06M	22.17M	0.48	23.75M	23.44M	1.32	243.02M	246.08M	1.24	23.75M	23.68M	0.31
16	22.15M	22.09M	0.26	23.90M	23.41M	2.09	243.19M	246.01M	1.14	23.90M	23.60M	1.26
17	22.24M	22.00M	1.07	24.04M	23.43M	2.62	243.34M	246.10M	1.12	24.04M	23.62M	1.76
18	22.71M	22.31M	1.79	24.34M	23.39M	4.05	243.58M	246.19M	1.06	24.34M	23.58M	3.22
19	22.82M	22.27M	2.48	24.53M	23.41M	4.79	243.78M	246.15M	0.96	24.53M	23.59M	3.95
20	23.17M	22.18M	4.45	24.71M	23.36M	5.79	243.98M	246.04M	0.84	24.71M	23.65M	4.50
25	24.10M	22.40M	7.59	25.94M	23.36M	11.03	245.19M	246.05M	0.35	25.94M	23.58M	10.01
30	25.55M	22.52M	13.46	27.52M	23.36M	17.81	246.51M	246.34M	0.07	27.52M	23.54M	16.91
35	26.83M	22.41M	19.74	28.48M	23.45M	21.46	247.85M	246.77M	0.44	28.48M	23.76M	19.88
40	28.88M	22.35M	29.23	30.54M	23.61M	29.35	249.66M	246.62M	1.23	30.54M	23.88M	27.89
46	31.30M	22.29M	40.39	31.87M	23.52M	35.51	252.02M	246.83M	2.10	31.87M	23.76M	34.11

The qualitative behavior of the model is exactly as the previous: very low errors up to 11 workers, when the Collector becomes the bottleneck. It is interesting to notice that the Collector is slightly slower than the Emitter. This is partially due to a higher value of k, but also because of the fact that the Collector *read and write* to the same memory interface. However, from the model point of view the only difference is that they have different values of R_q and thus different T_S times.

It is already interesting to note that this version perform better with slow parallelism degrees: the service time of each worker is $\sim 240M$ clocks w.r.t the $\sim 260M$ clocks of the previous one; the difference is, of course, approximately T_{copy}. So, the overlapping mechanism works well in this program, and is correctly approached in the Queueing Network model. As before, we apply the cutoff method to evaluate the service time when the bottleneck is moved. This way we obtain the model predictions of Table 8.10 and Figure 8.7.

Here the error is slightly higher, but still lower than $\sim 2.5\%$ in the pre-cutoff interval, and at most $\sim 4.35\%$ in general. This error, however, is not fully related to the estimation of R_q but, partially, also to some unexpected overheads in the parallel implementation: when the Collector is the bottleneck the service time of the farm T_S is not exactly T_{S-C} as expected, but is slightly higher. Thus, the overall error w.r.t our model (that is already underestimating T_{S-C}) is further increased. For example, if we consider the case with 11 workers, the error of T_{S-C} is $\sim 1.13\%$, while the error of T_S grows to $\sim 2.3\%$. Nevertheless, we consider the estimation quite accurate and sufficient for our purpose.

As an interesting side note, while the implementation is faster w.r.t the message passing in the first phase, when the Collector becomes the bottleneck, this version becomes slightly slower, because this Collector is slower than the Emitter of the other version.

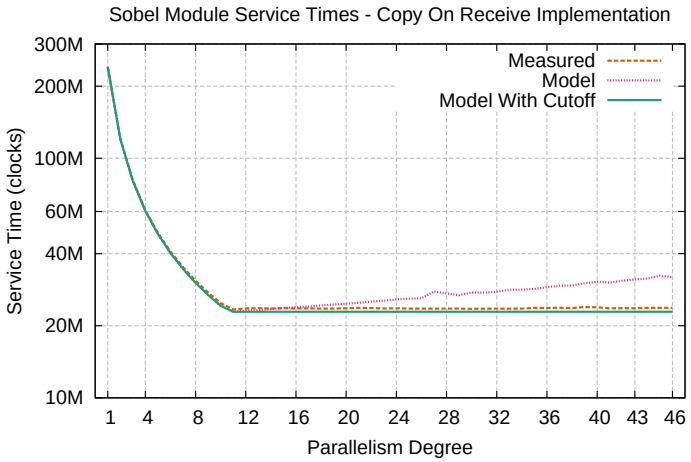


Figure 8.7: Comparison between architecture and model values of T_S for the copy on receive message passing imp. of the Sobel Module. Times in clock cycles.

Table 8.10: Comparison between architecture and final model values of T_S for the message passing implementation of the Sobel Module. Times in clock cycles.

Par. Degree	T_S Expected	T_S Measured	% Error	Par. Degree	T_S Expected	T_S Measured	% Error
1	240.56M	240.87M	0.13	24	22.88M	23.65M	3.27
2	120.35M	120.56M	0.17	25	22.88M	23.58M	2.95
3	80.27M	81.09M	1.01	26	22.88M	23.59M	3.02
4	60.30M	60.87M	0.94	27	22.88M	23.59M	2.99
5	48.24M	48.81M	1.18	28	22.88M	23.59M	3.02
6	40.19M	40.83M	1.57	29	22.88M	23.58M	2.99
7	34.50M	35.13M	1.78	30	22.88M	23.54M	2.82
8	30.20M	30.81M	1.98	31	22.88M	23.55M	2.83
9	26.85M	27.52M	2.45	32	22.88M	23.59M	3.01
10	24.20M	24.84M	2.57	33	22.88M	23.55M	2.85
11	22.88M	23.42M	2.31	34	22.88M	23.58M	2.95
12	22.88M	23.61M	3.08	35	22.88M	23.76M	3.69
13	22.88M	23.69M	3.43	36	22.88M	23.73M	3.57
14	22.88M	23.61M	3.10	37	22.88M	23.78M	3.80
15	22.88M	23.68M	3.38	38	22.88M	23.71M	3.50
16	22.88M	23.60M	3.05	39	22.88M	23.92M	4.35
17	22.88M	23.62M	3.15	40	22.88M	23.88M	4.19
18	22.88M	23.58M	2.97	41	22.88M	23.64M	3.23
19	22.88M	23.59M	3.03	42	22.88M	23.68M	3.38
20	22.88M	23.65M	3.26	43	22.88M	23.66M	3.29
21	22.88M	23.76M	3.69	44	22.88M	23.76M	3.71
22	22.88M	23.71M	3.52	45	22.88M	23.76M	3.71
23	22.88M	23.60M	3.06	46	22.88M	23.76M	3.71

8.5 Study of the pointer passing implementation

In this last implementation we completely remove the copies, as we exploit the shared memory architecture to exchange pointers between the pipeline stages that compose the farm. The performance model is now even simpler, as the Emitter and the Collector have a negligible service time:

$$\begin{aligned} T_{S-E} &= \sim 0 \\ T_{S-W_i} &= T_{W-calc} \\ T_{S-C} &= \sim 0 \end{aligned}$$

In addition, because of the pipeling effect of the scheme we have that the global service time of the farm can still be approximated as

$$T_{S-FARM} = \max\left\{T_{S-E}, \frac{T_{S-W_i}}{n}, T_{S-C}\right\}$$

The previous estimations of T_{W-calc} still holds, and we do not need T_{copy} anymore.

$$T_{w-calc} = 224896296 + (163180 \times Predicted\_Memory\_Latency)$$

However, as previously explained, we cannot use the **NUMA-like** allocation policy in this implementation because in the moment the input images are received by the Emitter, we cannot identify the correct worker. The most reasonable approach is to use an **SMP-like** allocation policy that, however, is known to perform worse, according to the study of Chapter 6, especially with a limited number of workers, because of the increased value of L_{net}.

8.5.1 Architecture Model Parameters

With this implementation we expect most of the parameters to be different, as we are introducing several novelties.

We still assume that the Emitter and the Collector are not a bottleneck, so that the whole system will be limited by the throughput of the Workers. In this implementation, however, this is a much more feasible assumption, as we have $T_{S-E} = T_{S-C} \simeq 0$: it is practically impossible for them to become the bottleneck of the farm. So we expect to not need to use the cutoff method to approximate the T_S, that will always be represented by the service time of the workers:

$$T_{S-FARM} = \max\left\{T_{S-E}, \frac{T_{S-W_i}}{n}, T_{S-C}\right\} = \frac{T_{S-W_i}}{n}$$

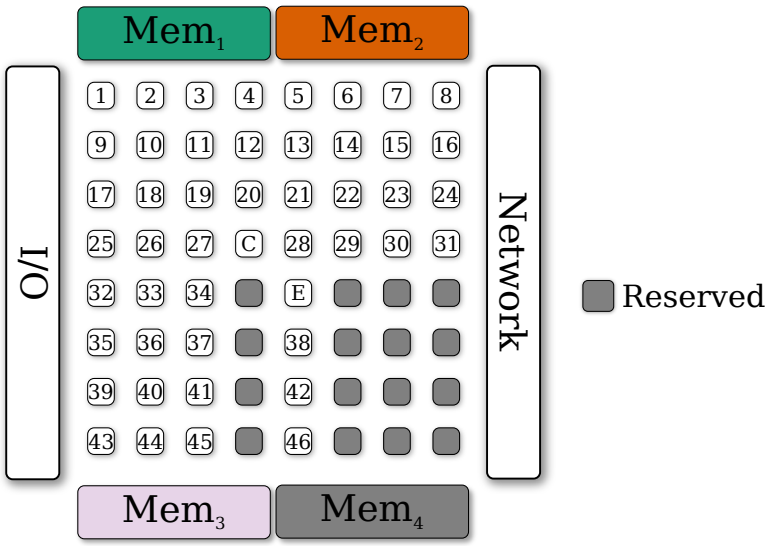


Figure 8.8: Allocation pattern for the processes in the pointer passing impl.

In this version, for a specific parallelism degree n we will have n processors that behave as workers, plus the Emitter and the Collector. The work of both the Emitter and the Collector, however, is negligible w.r.t the memory subsystem, so they will not be considered in the model.

Placement on the mesh

Given the use of the **SMP-like** memory policy, we also need to change the process allocation on the mesh. The study in Chapter 6 showed no real advantages in using a specific process allocation, so we selected the **linear**, that is simpler to be implemented. The new mapping scheme is depicted in Figure 8.8. Notice that the Emitter and the Collector are still at the center of the mesh, even if this is not required in this case.

Evaluating T_p

Given the absence (in the model) of the Emitter and the Collector, we just need to evaluate the T_p for the Workers. They just compute the tasks, without paying any additional costs, so we can estimate their service time as the one with the copy on receive implementation:

$$T_{p-W} = \frac{224896296}{(163180 + 159955)} = 695.98$$

Evaluating p_{mod}

The value for p_{mod} of T_{w-calc}, inherited from the sequential code, is still the same: $p_{mod}^{calc} = 0.5$ so, as in the copy on receive implementation, we have

$$p_{mod-W} = 0.5$$

Evaluating p_{mc}

For the estimation of p_{mc} we need to consider the fact that now the memory areas are allocated *differently*, with each buffer uniformly distributed among the three available memory controllers. This means that each worker will uniformly distribute its requests:

$$p_{mc1-W} = 0.33, p_{mc2-W} = 0.33, p_{mc3-W} = 0.33, p_{mc4-W} = 0$$

for every processor enabled in the model.

Evaluating k

The values for k, still obtained by executing the parallel implementation with parallelism degree of one, are:

$$k_{Wi} = 3$$

that is the same of the copy on receive implementation. This is not a case, as the two worker implementations are quite similar: another confirmation that k depends on the code to be executed.

Evaluating p_W

Given the new access pattern, we expect a different family of values for P_W.

We start with the reasoning of before, using a single worker: we have that, for each image, the worker will generate $163180 + 159955 + 160000$ requests for the calculation, of which 160000 correspond to write operations. The Emitter and the Collector are considered negligible. The requests will be uniformly distributed on the three memory controllers, so we have:

$$p_{W1} = \frac{160000}{3} \times \frac{3}{(163180 + 159955 + 160000)} = 0.33$$
$$p_{W2} = \frac{160000}{3} \times \frac{3}{(163180 + 159955 + 160000)} = 0.33$$
$$p_{W3} = \frac{160000}{3} \times \frac{3}{(163180 + 159955 + 160000)} = 0.33$$
$$p_{W4} = 0$$

When adding multiple workers, the amount of requests is increased, but the ratio is kept the same (i.e. with 2 workers we have 2× read and 2× write requests; with 3 workers 3× and so on).

Parameter Summary

For convenience, here we report all the parameters in a concise form. A notable aspect of this implementation is that, given the absence of the Emitter and the Collector, the parameters are the same for any parallelism degree. The only difference is given by the number of processes and their allocation on the processors.

$$
\begin{aligned}
T_{p-W} &= 695.98 \\
p_{mod-W} &= 0.5 \\
p_{mc1-W} &= 0.33 \\
p_{mc2-W} &= 0.33 \\
p_{mc3-W} &= 0.33 \\
p_{mc4-W} &= 0 \\
k_W &= 3 \\
p_{W1} &= 0.33 \\
p_{W2} &= 0.33 \\
p_{W3} &= 0.33 \\
p_{W4} &= 0
\end{aligned}
$$

8.5.2 Predicted Service Times

Now we can solve the queueing network system by using **EQNSim** and obtain the R_q values of interest. In particular, just need the average R_q of all the Workers, reported in Table 8.11. We used these values to obtain the prediction of T_{S-W_i} and T_S in Table 8.12.

# W	R_q	# W	R_q	# W	R_q	# W	R_q	# W	R_q
1	111.00	11	114.09	21	121.28	31	132.08	41	147.41
2	109.50	12	113.90	22	121.77	32	133.35	42	149.00
3	109.47	13	115.17	23	123.18	33	134.75	43	150.67
4	109.57	14	114.91	24	124.34	34	136.01	44	152.81
5	110.43	15	115.23	25	125.37	35	137.50	45	154.52
6	110.63	16	117.77	26	126.35	36	139.03	46	156.80
7	110.03	17	118.13	27	127.72	37	140.25		
8	111.28	18	119.08	28	128.71	38	141.80		
9	112.61	19	119.55	29	129.70	39	143.43		
10	113.01	20	120.88	30	130.73	40	145.48		

Table 8.11: R_q values obtained by solving the Queueing Network model parameterized for the pointer passing implementation. Times in clock cycles.

Table 8.12: Model values of T_{S-W_i} and T_S for the pointer passing imp., w.r.t the execution. Times in clock cycles.

# W	T_{S-W_i} Exp.	T_{S-W_i} Meas.	T_{S-W_i} % E.	T_S Exp.	T_S Meas.	T_S % E.	# W	T_{S-W_i} Exp.	T_{S-W_i} Meas.	T_{S-W_i} % E.r	T_S Exp.	T_S Meas.	T_S % E.
1	243.01M	242.38M	0.26	243.01M	242.47M	0.22	24	245.19M	248.57M	1.36	10.22M	10.39M	1.70
2	242.76M	242.52M	0.10	121.38M	121.35M	0.03	25	245.35M	248.44M	1.24	9.81M	9.96M	1.47
3	242.76M	242.57M	0.08	80.92M	81.05M	0.16	26	245.51M	249.22M	1.49	9.44M	9.63M	1.92
4	242.78M	242.75M	0.01	60.69M	60.74M	0.07	27	245.74M	249.49M	1.50	9.10M	9.28M	1.93
5	242.92M	243.14M	0.09	48.58M	48.71M	0.25	28	245.90M	249.63M	1.49	8.78M	8.95M	1.91
6	242.95M	243.33M	0.16	40.49M	40.65M	0.39	29	246.06M	249.87M	1.53	8.48M	8.66M	2.00
7	242.85M	243.61M	0.31	34.69M	34.91M	0.61	30	246.23M	249.87M	1.46	8.21M	8.36M	1.85
8	243.06M	243.80M	0.30	30.38M	30.52M	0.46	31	246.45M	250.50M	1.62	7.95M	8.12M	2.11
9	243.27M	244.37M	0.45	27.03M	27.24M	0.79	32	246.66M	250.72M	1.62	7.71M	7.87M	2.12
10	243.34M	244.33M	0.41	24.33M	24.49M	0.63	33	246.89M	250.79M	1.56	7.48M	7.64M	2.12
11	243.51M	244.94M	0.58	22.14M	22.32M	0.81	34	247.09M	251.18M	1.63	7.27M	7.43M	2.15
12	243.48M	245.23M	0.71	20.29M	20.47M	0.89	35	247.33M	251.38M	1.61	7.07M	7.23M	2.19
13	243.69M	245.73M	0.83	18.75M	18.94M	1.00	36	247.58M	251.74M	1.65	6.88M	7.03M	2.23
14	243.65M	245.85M	0.90	17.40M	17.61M	1.19	37	247.78M	251.99M	1.67	6.70M	6.85M	2.24
15	243.70M	246.03M	0.95	16.25M	16.43M	1.14	38	248.03M	252.23M	1.66	6.53M	6.68M	2.23
16	244.11M	246.43M	0.94	15.26M	15.44M	1.20	39	248.30M	252.59M	1.70	6.37M	6.52M	2.35
17	244.17M	246.84M	1.08	14.36M	14.57M	1.43	40	248.64M	252.30M	1.45	6.22M	6.34M	1.97
18	244.33M	247.07M	1.11	13.57M	13.78M	1.49	41	248.95M	253.03M	1.61	6.07M	6.21M	2.19
19	244.41M	247.42M	1.22	12.86M	13.07M	1.58	42	249.21M	253.31M	1.62	5.93M	6.07M	2.19
20	244.62M	247.25M	1.06	12.23M	12.41M	1.41	43	249.48M	253.46M	1.57	5.80M	5.93M	2.15
21	244.69M	247.92M	1.31	11.65M	11.85M	1.67	44	249.83M	253.86M	1.59	5.68M	5.80M	2.13
22	244.77M	248.12M	1.35	11.13M	11.33M	1.79	45	250.11M	254.04M	1.55	5.56M	5.68M	2.11
23	245.00M	248.29M	1.33	10.65M	10.85M	1.80	46	250.48M	254.46M	1.56	5.45M	5.56M	2.14

In this case the model already approximate the behavior for the whole range of parallelism degrees. Without having the problem of Emitters and Collectors, now the farm is able to further improve the service times, that continue to decrease throughout the whole curve (as depicted in Figure 8.9).

This version is surely performing better than the others with a large number of cores, being able to achieve a T_S with 46 workers $\sim 4\times$ lower than the previous ones. However, with a limited number of workers, the times of the message passing with copy on receive seems to be slightly better. This is motivated by the different memory allocation policy, that causes lower R_q in the message passing implementations: with **Copy on Receive** the single worker run have an R_q of only 96 clocks, compared with the 111 of this version. However, this difference decrease when the number of workers grows, because in the pointer passing implementation we completely avoid the copies that, of course, increase the pressure on the memory subsystem. Thus, having these two adverse effects, it is not straightforward to decide whether implementation will perform better: only the application of the Queueing Network model allow us to estimate the R_q and thus select the best farm.

The overall error on T_S is actually very low, always under $\sim 2.35\%$, so even in this case our model was able to predict the performance of the parallel implementation.

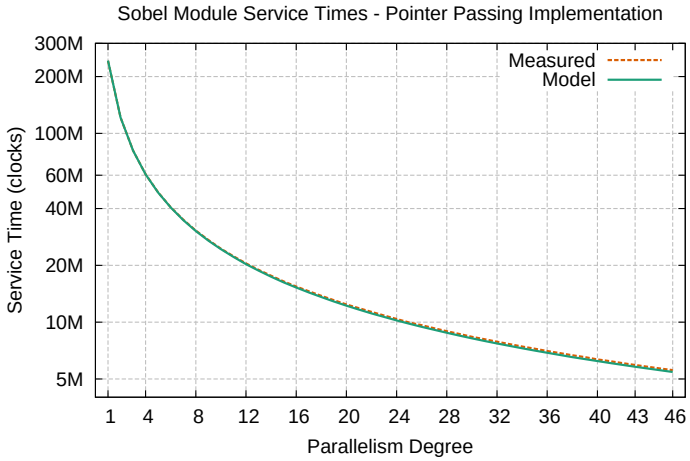


Figure 8.9: Comparison between architecture and model values of T_S for pointer passing implementation of the Sobel Module. Times in clock cycles.

8.6 Selection of the best implementation

Now that we have modeled the three implementations, and verified experimentally the accuracy of the modeled T_S for each of them, we can obtain the service time function $T_s^{(n)}$ of the module in which, for each parallelism degree, we select the best implementation. This is the function that will be used by the compiler in solving the parallel application graph to select, for each module, the correct parallelism

degree (and mapping on the architectures), as discussed in Chapter 4. Given the accuracy of the single service times, we expect very good results here. In Table 8.13 we summarize the service times for each version, and show the $T_s^{(n)}$ of the pattern, along with the selected implementation for each parallelism degree.

The results are graphically shown in Figure 8.10. For the sake of readability we do not plot the service times directly, but the speedup of the parallel version w.r.t the sequential code, defined as follows:

$$S^{(n)} = \frac{T_{S-seq}}{T_{S-par}^{(n)}}$$

The results clearly shows that the simple message passing implementation is not very effective, as it is never faster than the others. In fact, requiring the worker to execute the message copy increase its service times; the other implementations avoid this problem without adding notable overheads, so they are always faster.

The comparison between message passing with copy on receive and pointer passing basically shows the behavior that we expected, but in a very limited scale: with a small number of cores message passing perform better because of the **NUMA-like** allocation. In fact, according to the model, it performs better **as long as the Collector is not the bottleneck**: this would be indeed an interesting result if confirmed with other applications. However, if we analyze the performance improvement of the copy on receive w.r.t the pointer passing with the same parallelism degree, we see that we are talking of a very limited difference, within the 1%. This is due to the limited distance of the memory controllers in the Tile*Pro*64 architecture: as already explained at the beginning of Chapter 6, in this architecture the l_{net} difference between a "local" and a "remote" memory is just of $\sim 19\%$, because they are all inside the same chip. Using the same implementations on other multi-core architectures, such as the Intel or AMD multi-chip configurations, will probably produce a much more interesting gain, because the "remote" memory have a l_{net} almost doubled w.r.t the local one.

Nevertheless, *in this case*, the advantage of the message passing implementation with proper memory allocation is very small. Considering that our model predict the service time with an approximation within $\sim 2-4\%$, higher than the difference of the two implementation in the $[1, 10]$ area, we *could* have made the wrong decision (i.e. maybe the pointer passing implementation was better). On the other hand. we have no doubts in the $[11, 48]$ area because the pointer passing implementation is always better of *at least* $\sim 4\%$.

Table 8.13: Comparison of the various implementation service times, according to the model. T_S for the best choice and improvement w.r.t the 2^{nd}. Times in clock cycles. PTR: Pointer Passing, MSG: Message Passing, COR: Copy on Receive.

#	PTR	MSG	COR	Best	Sel.	%Impr.	#	PTR	MSG	COR	Best	Sel.	%Impr.
1	243,01M	262,06M	240,56M	240,56M	COR	1,02	24	10,22M	22,05M	22,88M	10,22M	PTR	123,96
2	121,38M	131,14M	120,35M	120,35M	COR	0,85	25	9,81M	22,05M	22,88M	9,81M	PTR	133,13
3	80,92M	87,48M	80,27M	80,27M	COR	0,82	26	9,44M	22,05M	22,88M	9,44M	PTR	142,30
4	60,69M	65,67M	60,30M	60,30M	COR	0,65	27	9,10M	22,05M	22,88M	9,10M	PTR	151,39
5	48,58M	52,56M	48,24M	48,24M	COR	0,71	28	8,78M	22,05M	22,88M	8,78M	PTR	160,53
6	40,49M	43,81M	40,19M	40,19M	COR	0,75	29	8,48M	22,05M	22,88M	8,48M	PTR	169,66
7	34,69M	37,67M	34,50M	34,50M	COR	0,56	30	8,21M	22,05M	22,88M	8,21M	PTR	178,76
8	30,38M	32,97M	30,20M	30,20M	COR	0,62	31	7,95M	22,05M	22,88M	7,95M	PTR	187,80
9	27,03M	29,38M	26,85M	26,85M	COR	0,67	32	7,71M	22,05M	22,88M	7,71M	PTR	196,83
10	24,33M	26,49M	24,20M	24,20M	COR	0,53	33	7,48M	22,05M	22,88M	7,48M	PTR	205,82
11	22,14M	24,12M	22,88M	22,14M	PTR	3,35	34	7,27M	22,05M	22,88M	7,27M	PTR	214,83
12	20,29M	22,16M	22,88M	20,29M	PTR	12,76	35	7,07M	22,05M	22,88M	7,07M	PTR	223,77
13	18,75M	22,05M	22,88M	18,75M	PTR	22,06	36	6,88M	22,05M	22,88M	6,88M	PTR	232,68
14	17,40M	22,05M	22,88M	17,40M	PTR	31,47	37	6,70M	22,05M	22,88M	6,70M	PTR	241,65
15	16,25M	22,05M	22,88M	16,25M	PTR	40,83	38	6,53M	22,05M	22,88M	6,53M	PTR	250,53
16	15,26M	22,05M	22,88M	15,26M	PTR	49,96	39	6,37M	22,05M	22,88M	6,37M	PTR	259,37
17	14,36M	22,05M	22,88M	14,36M	PTR	59,30	40	6,22M	22,05M	22,88M	6,22M	PTR	268,09
18	13,57M	22,05M	22,88M	13,57M	PTR	68,56	41	6,07M	22,05M	22,88M	6,07M	PTR	276,81
19	12,86M	22,05M	22,88M	12,86M	PTR	77,87	42	5,93M	22,05M	22,88M	5,93M	PTR	285,60
20	12,23M	22,05M	22,88M	12,23M	PTR	87,06	43	5,80M	22,05M	22,88M	5,80M	PTR	294,35
21	11,65M	22,05M	22,88M	11,65M	PTR	96,36	44	5,68M	22,05M	22,88M	5,68M	PTR	302,96
22	11,13M	22,05M	22,88M	11,13M	PTR	105,65	45	5,56M	22,05M	22,88M	5,56M	PTR	311,65
23	10,65M	22,05M	22,88M	10,65M	PTR	114,79	46	5,45M	22,05M	22,88M	5,45M	PTR	320,18

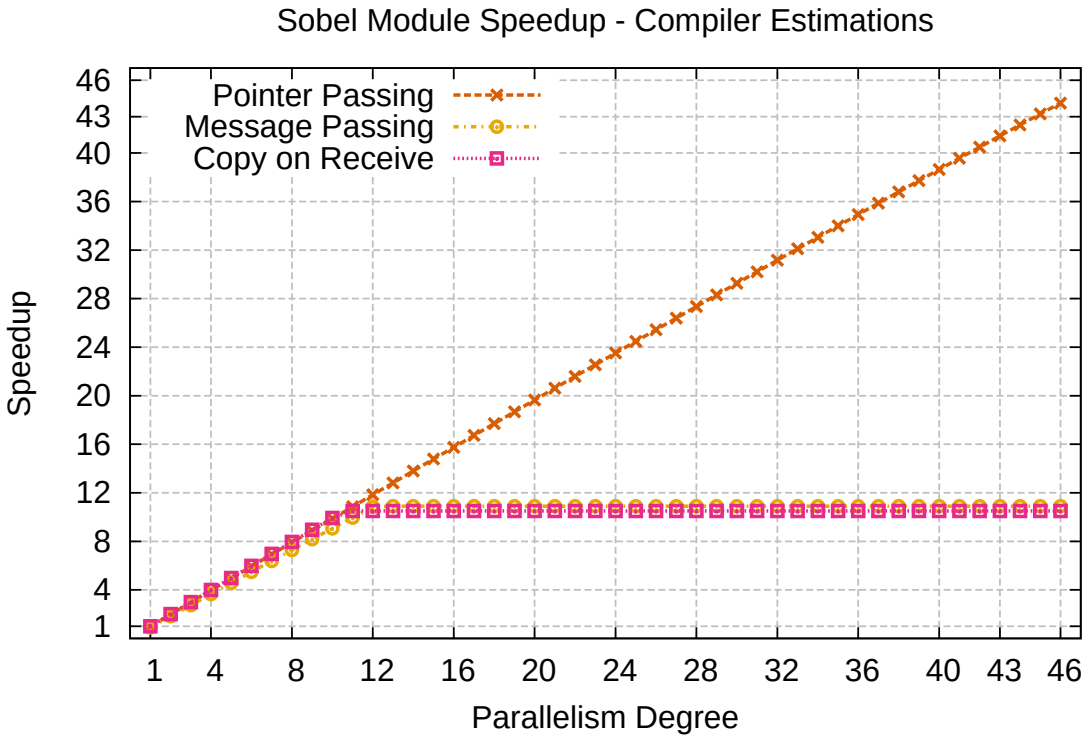


Figure 8.10: Comparison between measured values of the speedup of the module.

We compare the modeled $T_s^{(n)}$ with the real implementations in Table 8.14 (and in Figure 8.11 using the speedup metric). As expected, the error in the estimation of $T_s^{(n)}$ is very good, always within the $\sim 2.4\%$. Thus, our estimation of the service time can be considered sufficiently good to let the compiler select the proper parallelism degree to remove the bottleneck on the graph of modules.

Table 8.14 also show that the compiler was not able to correctly predict the best implementation for the $[3, 10]$ range: we selected the copy on receive message passing implementation, while the "numerical" best is the pointer passing one. So, in this cases, the compiler would have (in theory) chosen the wrong implementation, but in practice the performance loss can be considered negligible: for the sake of completeness we report in Table 8.15 the differences in the service times of the implementations run on the Tile*Pro*64 architecture by using the best choice for the model and the real best implementation. We see that the performance loss is lower than 1.5%, thus we consider the compiler choice very good.

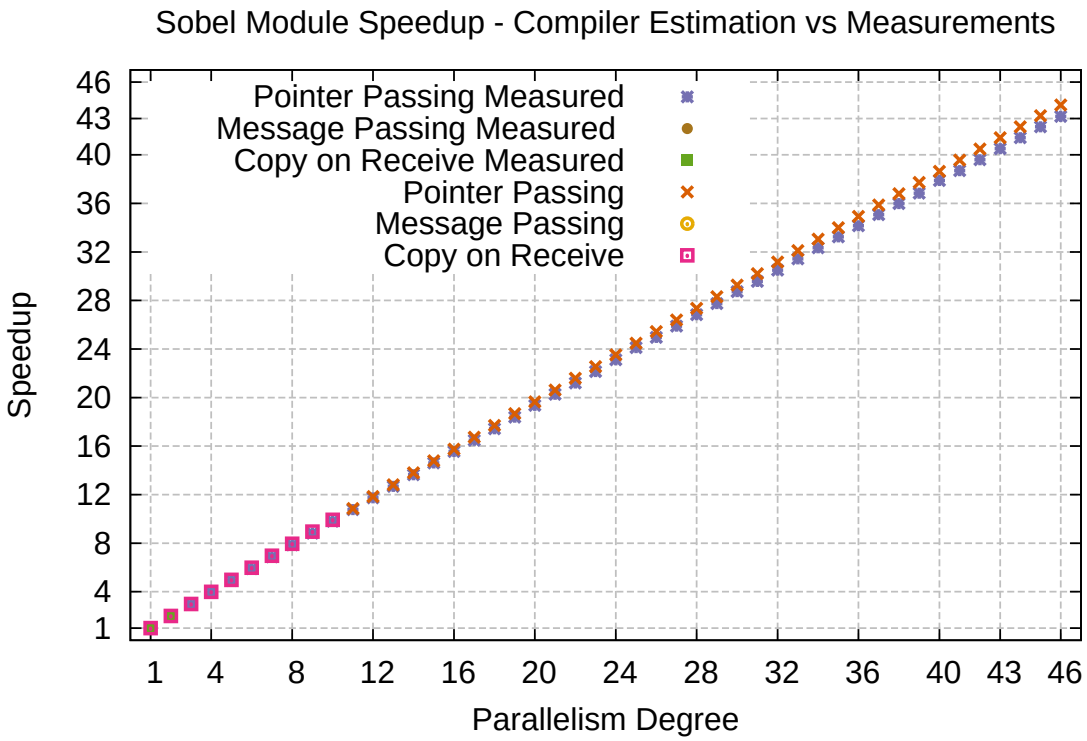


Figure 8.11: Comparison between measured and model values of the speedup of the Sobel parallel module.

Table 8.14: Comparison of the predicted service time function w.r.t the measured one, highlighting the implementation chosen in both cases. Times in clock cycles. PTR: Pointer Passing, COR: Copy on Receive.

#	Modeled		Measured		%	#	Modeled		Measured		%
W	T_S	Impl.	T_S	Impl.	Error	W	T_S	Impl.	T_S	Impl.	Error
1	240,56M	COR	240,87M	COR	0,127518	24	10,22M	PTR	10,39M	PTR	1,732916
2	120,35M	COR	120,56M	COR	0,167644	25	9,81M	PTR	9,96M	PTR	1,494195
3	80,27M	COR	81,05M	PTR	0,977131	26	9,44M	PTR	9,63M	PTR	1,957347
4	60,30M	COR	60,74M	PTR	0,723249	27	9,10M	PTR	9,28M	PTR	1,97213
5	48,24M	COR	48,71M	PTR	0,970666	28	8,78M	PTR	8,95M	PTR	1,94474
6	40,19M	COR	40,65M	PTR	1,136708	29	8,48M	PTR	8,66M	PTR	2,04215
7	34,50M	COR	34,91M	PTR	1,173815	30	8,21M	PTR	8,36M	PTR	1,884004
8	30,20M	COR	30,52M	PTR	1,081506	31	7,95M	PTR	8,12M	PTR	2,159533
9	26,85M	COR	27,24M	PTR	1,465257	32	7,71M	PTR	7,87M	PTR	2,161441
10	24,20M	COR	24,49M	PTR	1,17173	33	7,48M	PTR	7,64M	PTR	2,163588
11	22,14M	PTR	22,32M	PTR	0,813388	34	7,27M	PTR	7,43M	PTR	2,195764
12	20,29M	PTR	20,47M	PTR	0,893607	35	7,07M	PTR	7,23M	PTR	2,241263
13	18,75M	PTR	18,94M	PTR	1,014783	36	6,88M	PTR	7,03M	PTR	2,281728
14	17,40M	PTR	17,61M	PTR	1,205768	37	6,70M	PTR	6,85M	PTR	2,286799
15	16,25M	PTR	16,43M	PTR	1,157103	38	6,53M	PTR	6,68M	PTR	2,277581
16	15,26M	PTR	15,44M	PTR	1,21968	39	6,37M	PTR	6,52M	PTR	2,406686
17	14,36M	PTR	14,57M	PTR	1,452016	40	6,22M	PTR	6,34M	PTR	2,011089
18	13,57M	PTR	13,78M	PTR	1,516764	41	6,07M	PTR	6,21M	PTR	2,243848
19	12,86M	PTR	13,07M	PTR	1,604209	42	5,93M	PTR	6,07M	PTR	2,239672
20	12,23M	PTR	12,41M	PTR	1,43369	43	5,80M	PTR	5,93M	PTR	2,193459
21	11,65M	PTR	11,85M	PTR	1,697899	44	5,68M	PTR	5,80M	PTR	2,181181
22	11,13M	PTR	11,33M	PTR	1,82449	45	5,56M	PTR	5,68M	PTR	2,15403
23	10,65M	PTR	10,85M	PTR	1,836107	46	5,45M	PTR	5,56M	PTR	2,185111

Table 8.15: Comparison the service time function using the real best implementation w.r.t the best according to the model. Times in clock cycles. PTR: Pointer Passing, COR: Copy on Receive.

#	Best for Model		Best Impl.		%	#	Best for Model		Best Impl.		%
W	T_S	Impl.	T_S	Impl.	Error	W	T_S	Impl.	T_S	Impl.	Error
1	240.87M	COR	240.87M	COR	0	24	10.39M	PTR	10.39M	PTR	0
2	120.56M	COR	120.56M	COR	0	25	9.96M	PTR	9.96M	PTR	0
3	81.09M	COR	81.05M	PTR	0.047753	26	9.63M	PTR	9.63M	PTR	0
4	60.87M	COR	60.74M	PTR	0.225205	27	9.28M	PTR	9.28M	PTR	0
5	48.81M	COR	48.71M	PTR	0.220675	28	8.95M	PTR	8.95M	PTR	0
6	40.83M	COR	40.65M	PTR	0.455734	29	8.66M	PTR	8.66M	PTR	0
7	35.13M	COR	34.91M	PTR	0.63024	30	8.36M	PTR	8.36M	PTR	0
8	30.81M	COR	30.52M	PTR	0.932413	31	8.12M	PTR	8.12M	PTR	0
9	27.52M	COR	27.24M	PTR	1.026677	32	7.87M	PTR	7.87M	PTR	0
10	24.84M	COR	24.49M	PTR	1.44929	33	7.64M	PTR	7.64M	PTR	0
11	22.32M	PTR	22.32M	PTR	0	34	7.43M	PTR	7.43M	PTR	0
12	20.47M	PTR	20.47M	PTR	0	35	7.23M	PTR	7.23M	PTR	0
13	18.94M	PTR	18.94M	PTR	0	36	7.03M	PTR	7.03M	PTR	0
14	17.61M	PTR	17.61M	PTR	0	37	6.85M	PTR	6.85M	PTR	0
15	16.43M	PTR	16.43M	PTR	0	38	6.68M	PTR	6.68M	PTR	0
16	15.44M	PTR	15.44M	PTR	0	39	6.52M	PTR	6.52M	PTR	0
17	14.57M	PTR	14.57M	PTR	0	40	6.34M	PTR	6.34M	PTR	0
18	13.78M	PTR	13.78M	PTR	0	41	6.21M	PTR	6.21M	PTR	0
19	13.07M	PTR	13.07M	PTR	0	42	6.07M	PTR	6.07M	PTR	0
20	12.41M	PTR	12.41M	PTR	0	43	5.93M	PTR	5.93M	PTR	0
21	11.85M	PTR	11.85M	PTR	0	44	5.80M	PTR	5.80M	PTR	0
22	11.33M	PTR	11.33M	PTR	0	45	5.68M	PTR	5.68M	PTR	0
23	10.85M	PTR	10.85M	PTR	0	46	5.56M	PTR	5.56M	PTR	0

8.7 Impact of a multi-chip configuration

The study of the three implementations showed an extremely good estimation on the service times, proving that it is feasible to use the methodology presented in this thesis to compare and select different implementations of a parallel module. However, from a qualitative point of view, the application of the model showed that, in practice, the pointer passing implementation perform always better or as good as the others.

This result is mainly because of the limited advantage provided by the **NUMA-like RR** allocation policy w.r.t the **SMP-like** policy used with pointer passing. This is related to the target architecture, specifically because all the memory controllers are *on the same chip*, and thus the cost of accessing a "remote" memory is only slightly higher than accessing the "local" one. As discussed at the beginning of Chapter 6, the difference between local and remote memories is much more evident in the case of multi-chip configurations, that constitute the majority of commercial parallel machines.

Unfortunately, our architecture model has been developed for the Tilera Tile*Pro*64 architecture, that does not allow this configuration. Yet, we can "imagine" a multi-chip Tile*Pro*64 server and use the model - adequately extended - to study the behavior of our Sobel module on a hypothetical multi-chip configuration.

8.7.1 A multi-chip Tile*Pro*64 configuration

The Tilera Tile*Pro*64 processor does in fact allow multi-chip configurations; however, this can be done by connecting different Tile*Pro*64 using the PCIe connection (the same used to connect the processor to a host system when used as accelerator). Unfortunately this does not allow the use of a unified physical address space, and thus result in a very different approach w.r.t the other commercial solutions. We therefore *imagine* a specific version of the Tile*Pro*64 in which multiple chip can be connected and share the physical address space. Technically, this would require a different routing policy on the mesh, to forward "off-chip" request to/from the external connection interface. To preserve the bandwidth requirements, we use multiple mesh switch to connect the two chips, as depicted in Figure 8.12. By connecting an entire row of switch we are able to ensure a bandwidth of

$$27Gbps * 8 \simeq 216Gbps \tag{8.7}$$

compared with the nominal maximum bandwidth of the remote memory subsystem of

$$MaxBW_{MemCtl} * 4 \simeq 52Gbps * 4 = 208Gpbs \tag{8.8}$$

We assume our chip-to-chip connection to be able to provide this bandwidth; while this seems relatively high, we mention that the IBM Power7 chip, according to the specification[102], uses a chip-to-chip connection with a bandwidth of $\sim 360GB/s =$

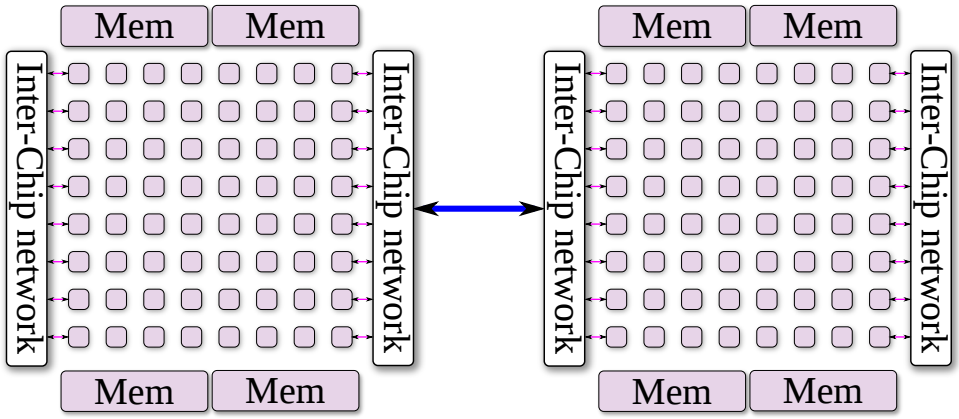


Figure 8.12: The multi-chip Tile*Pro*64 configuration we imagined for this evaluation.

2880*Gbps*, about an order of magnitude higher than the one we are requiring. For the latency values of the connection, we use the one measured in [131] for the QuickPath interconnection of the current Intel Xeon processors.

$$L_{chip-to-chip} = L_{QPI} = 41ns = 48clocks \tag{8.9}$$

Using this configuration, we are sure that the chip-to-chip connection will not become a bottleneck, and thus we can simply adapt our model by increasing the values of L_{net} correspondingly. We are aware that the end result is still a hypothetical architecture; yet, by using concepts and performance data already exploited in other architectures, we believe we imagined a *realistic architecture*.

8.7.2 Network Latencies

The Architectural model is unchanged, apart from the fact that we have doubled the processors and the memory interfaces. The only difference reside in the values of L_{net}. Routing on the mesh can be easily modified by considering a "larger" $16x8$ mesh, in such way that the XY routing protocol is maintained and that on-chip memories inherit the previous latencies. An off-chip request will just reach a border switch, will be forwarded to the other chip and then will continue its path towards the memory.

In Figure 8.13 we exemplify the forward/backward path for an off-chip memory request. For the sake of conciseness we just report the example L_{net} for core $< 0, 0 >$; the others are calculated accordingly.

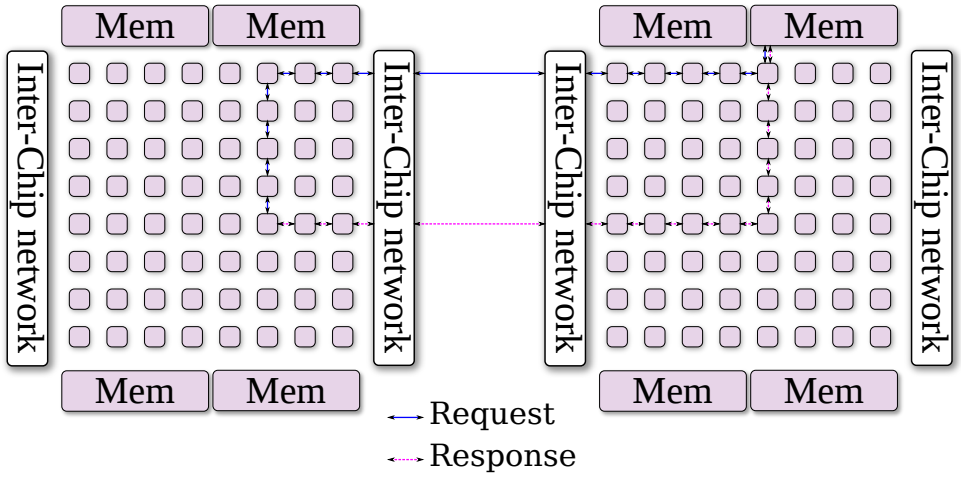


Figure 8.13: The path for a memory request on the multi-chip Tile*Pro*64 config.

$$
\begin{aligned}
L_{net}^{1} &= 22clocks \\
L_{net}^{2} &= 28clocks \\
L_{net}^{3} &= 36clocks \\
L_{net}^{4} &= 42clocks \\
L_{net}^{5} &= 86clocks \\
L_{net}^{6} &= 92clocks \\
L_{net}^{7} &= 100clocks \\
L_{net}^{8} &= 106clocks \\
L_{net}^{avg} &= 64clocks
\end{aligned}
$$

We can easily see the increase in the latency related to the change of chip, that produces L_{net} more than doubled when changing chip. Of course, on the overall R_q the increase is lower because the memory access time is kept constant ($R_q = L_{net} + L_{mem}$).

8.7.3 Core reservation and placement on the mesh

We adopt the same concept of the previous experiment, and reserve a part of the chip for other modules of the applications. In this case we decided to reserve one fourth of each chip. In this scenario we need to find two cores to act as Emitter and Collector. For the message passing implementation we should select the two in such a way that L_{net}^{avg}, i.e. the average distance to the memories, is minimized, given the impact of these entities in the parallel implementation. On the other hand, for the pointer passing implementation, because of their negligible effect, we select two

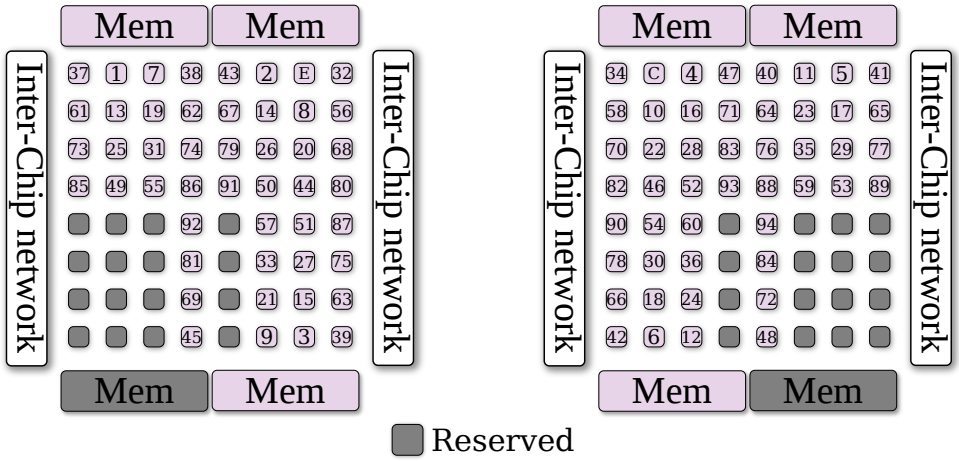


Figure 8.14: Process allocation in the multi-chip message passing implementation.

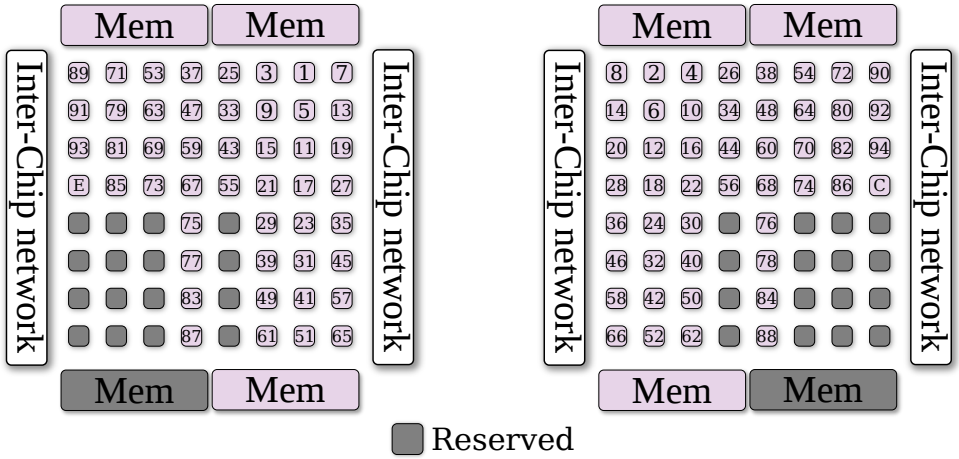


Figure 8.15: Process allocation in the multi-chip pointer passing implementation.

locations that *maximize* L_{net}^{avg}, so that the "fastest" cores are used for the workers. The final allocation is depicted in Figure 8.14 for message passing and Figure 8.15 for pointer passing.

8.7.4 Implementations and model parameters

For this study we neglect the original "message passing" implementation as we saw that it is always outperformed by the others. The comparison is therefore limited on two cases:

- **Pointer Passing**, that is usually the best choice of the studied implementations.

- **Message Passing with Copy on Receive** that, as we already saw, is able to compete, and in some cases outperform, the other because of its Numa-like memory allocation.

Pointer passing

In this case the model parameters are mostly the same as before, except for the fact that the memory requests are uniformly distributed among six memory interfaces:

$$p_{mc1-W} = 0.166, p_{mc2-W} = 0.166, p_{mc3-W} = 0, p_{mc4-W} = 0.166$$
$$p_{mc5-W} = 0.166, p_{mc6-W} = 0.166, p_{mc7-W} = 0.166, p_{mc8-W} = 0$$

for every processor enabled in the model.

Message Passing with Copy on Receive

With the message passing implementation all the memory-related parameters must be re-evaluated, using the same concepts as before. We just show the case of using 6 workers, as an example:

$$p_{mc1-E^{(6)}} = 0.0833, p_{mc2-E^{(6)}} = 0.5833, p_{mc3-E^{(6)}} = 0, p_{mc4-E^{(6)}} = 0.0833$$
$$p_{mc5-E^{(6)}} = 0.0833, p_{mc6-E^{(6)}} = 0.0833, p_{mc7-E^{(6)}} = 0.0833, p_{mc8-E^{(6)}} = 0$$
$$p_{mc1-C^{(6)}} = 0.1666, p_{mc2-C^{(6)}} = 0.1666, p_{mc3-C^{(6)}} = 0, p_{mc4-C^{(6)}} = 0.1666$$
$$p_{mc5-C^{(6)}} = 0.1666, p_{mc6-C^{(6)}} = 0.1666, p_{mc7-C^{(6)}} = 0.1666, p_{mc8-C^{(6)}} = 0$$
$$p_{W1^{(6)}} = p_{W4^{(6)}} = p_{W5^{(6)}} = p_{W6^{(6)}} = p_{W7^{(6)}} = 0.499$$
$$p_{W2^{(6)}} = 0.2502$$

The second memory controller have significantly different values because, being the "local" memory of the Emitter, receives all the read memory requests of the Emitter copy.

8.7.5 Performance study

We evaluated the Queueing Network models of the two implementations and obtained the values of R_q reported in Table 8.16, for the Pointer Passing one, and 8.17 for the other. The results already shows a sensibly different R_q for the workers in the two cases, that should translate in a better service time for the Copy On Receive implementation, as long as the Emitter or the Collector are not bottlenecks.

#W	R_q	#W	R_q	#W	R_q	#W	R_q
1	130.00	25	136.80	49	147.04	73	162.35
2	130.00	26	136.72	50	147.86	74	163.02
3	130.00	27	138.29	51	148.39	75	163.90
4	130.48	28	137.60	52	148.77	76	164.56
5	130.70	29	138.30	53	149.41	77	165.54
6	130.83	30	138.26	54	149.90	78	166.16
7	131.79	31	139.04	55	150.53	79	167.00
8	130.98	32	139.03	56	151.19	80	167.80
9	131.50	33	139.40	57	151.78	81	168.66
10	131.57	34	140.22	58	152.20	82	169.61
11	132.86	35	141.01	59	152.91	83	162.45
12	132.78	36	140.96	60	153.73	84	171.49
13	133.02	37	141.07	61	154.20	85	172.33
14	132.85	38	141.97	62	154.74	86	173.30
15	133.28	39	142.24	63	155.43	87	174.31
16	133.69	40	142.74	64	156.03	88	175.08
17	134.38	41	143.78	65	156.61	89	176.53
18	134.56	42	143.75	66	157.26	90	177.22
19	134.43	43	144.31	67	158.01	91	178.42
20	134.84	44	144.60	68	158.42	92	179.38
21	135.22	45	145.50	69	159.32	93	179.94
22	135.56	46	145.79	70	160.20	94	181.47
23	136.75	47	146.08	71	160.76		
24	136.49	48	146.49	72	161.57		

Table 8.16: R_q values obtained by solving the multi-chip Queueing Network model parameterized for the pointer passing implementation. Times in clock cycles.

# W	R_q Work.	Em.	Coll.	# W	R_q Work.	Em.	Coll.	# W	R_q Work.	Em.	Coll.	# W	R_q Work.	Em.	Coll.
1	97.94	147.65	160.62	25	107.11	156.90	165.28	49	117.35	171.78	179.42	73	109.28	186.71	191.35
2	98.17	148.85	156.96	26	108.34	158.93	166.18	50	117.18	173.07	180.60	74	109.69	188.89	193.68
3	99.24	148.60	157.72	27	109.25	159.26	167.14	51	116.30	173.97	180.54	75	109.87	188.55	194.55
4	97.90	148.42	157.41	28	109.69	159.62	167.74	52	115.70	174.36	181.60	76	109.80	188.16	194.92
5	98.58	148.76	157.39	29	109.78	159.38	166.72	53	114.84	174.50	181.13	77	109.80	188.97	194.71
6	98.37	148.61	157.01	30	109.84	160.13	167.94	54	114.13	175.06	182.45	78	109.50	189.59	195.69
7	97.86	149.18	157.06	31	110.24	160.42	168.52	55	113.40	174.77	181.38	79	109.97	189.59	196.01
8	99.26	149.86	158.86	32	111.81	162.39	169.70	56	113.31	176.97	183.49	80	110.13	192.33	198.10
9	100.35	149.97	158.82	33	112.61	162.89	170.38	57	113.28	177.56	184.21	81	110.71	193.83	198.94
10	100.67	150.85	159.10	34	112.96	163.14	170.60	58	112.76	177.11	183.36	82	110.19	191.11	198.23
11	100.06	150.74	158.58	35	113.19	163.59	171.28	59	111.92	178.13	184.45	83	110.62	195.81	199.08
12	101.27	151.01	159.07	36	113.62	163.40	171.52	60	111.39	177.81	185.13	84	110.50	192.81	201.35
13	100.70	150.21	159.94	37	114.01	163.57	171.78	61	110.89	177.94	185.92	85	110.57	193.63	199.69
14	102.16	152.14	160.76	38	115.29	166.27	172.83	62	111.23	180.32	186.59	86	111.09	196.74	201.37
15	103.48	152.99	162.02	39	115.42	166.83	174.20	63	111.16	181.27	187.46	87	111.69	199.32	201.94
16	102.98	153.51	161.44	40	116.56	166.80	175.02	64	110.99	173.50	188.23	88	111.54	197.95	203.17
17	103.26	153.51	162.12	41	117.09	167.24	175.00	65	110.42	181.34	186.39	89	111.90	198.39	202.64
18	103.10	153.61	161.58	42	117.32	167.42	175.19	66	110.04	182.03	188.32	90	111.71	198.49	203.17
19	104.49	153.70	162.24	43	118.35	167.81	175.88	67	109.76	182.71	188.41	91	111.87	197.48	203.24
20	105.53	155.51	163.35	44	119.06	169.07	178.02	68	109.92	184.16	189.66	92	112.97	201.26	206.20
21	106.58	156.07	164.22	45	120.30	171.17	178.17	69	110.40	181.09	190.42	93	113.68	202.88	207.22
22	105.34	156.25	164.16	46	120.40	171.70	178.30	70	109.84	184.77	191.05	94	113.61	201.52	207.07
23	107.05	156.27	164.75	47	119.45	171.88	179.08	71	109.85	186.13	190.61				
24	106.53	156.42	164.72	48	118.30	171.50	178.84	72	109.41	185.83	190.13				

Table 8.17: R_q values obtained by solving the multi-chip Queueing Network model parameterized for the copy on receive message passing implementation. Times in clock cycles.

Using these response times we are able to predict the average service times of the module. In Table 8.18 and Figure 8.16 we summarize the results. The behavior is much clearer this time: up to 10 workers the Message Passing with Copy on Receive outperform the other implementation of a fixed $\sim 2.1\%$. It is also interesting to note that in this part the service time is basically the same of the single-chip configuration: by using the NUMA allocation policy, we are not paying the cost of inter-chip communications. With more than 10 workers, as seen before, the Emitter/Collector become bottleneck, and of course increasing the parallelism degree does not improve the performances. In this area the pointer passing (characterized by the absence of heavyweight tasks in Emitter and Collector) is still providing the best result. However, the advantage in the first part is clear, and suggest that further optimized message-passing implementations may be interesting and provide even better results. For example, as demonstrated in one of our paper using the Intel Xeon PHI multiprocessor[47], it is possible to exploit hardware multithreading or multiple cores to improve the throughput of Emitter/Collectors. This way we could, for example, try to effectively overcome the 10-workers limitation and provide the $\sim 2.1\%$ performance improvement to a broader set of parallelism degrees.

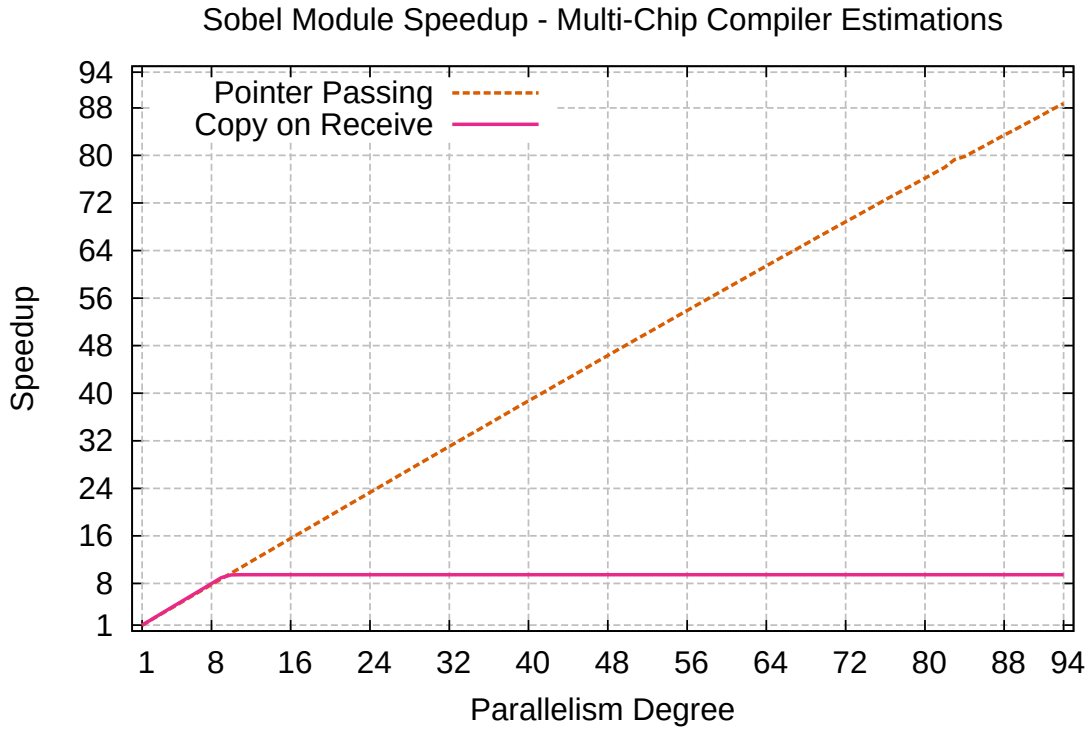


Figure 8.16: Comparison between estimated values of the speedup of the module in a multi-chip configuration.

Table 8.18: Comparison of the implementations, according to the multi-chip model. T_S for the best choice and improvement w.r.t the Pointer Passing implementation. Times in clock cycles. PTR: Pointer Passing, COR: Copy on Receive.

#	Best	Sel.	%Impr.	#	Best	Sel.	%Impr.	#	Best	Sel.	%Impr.	#	Best	Sel.	%Impr.
1	240.88M	COR	2.13	25	9.89M	PTR		49	5.08M	PTR		73	3.44M	PTR	
2	120.46M	COR	2.11	26	9.51M	PTR		50	4.98M	PTR		74	3.40M	PTR	
3	80.36M	COR	2.04	27	9.17M	PTR		51	4.88M	PTR		75	3.36M	PTR	
4	60.22M	COR	2.16	28	8.83M	PTR		52	4.79M	PTR		76	3.31M	PTR	
5	48.20M	COR	2.13	29	8.53M	PTR		53	4.70M	PTR		77	3.27M	PTR	
6	40.16M	COR	2.15	30	8.25M	PTR		54	4.62M	PTR		78	3.23M	PTR	
7	34.41M	COR	2.25	31	7.99M	PTR		55	4.54M	PTR		79	3.19M	PTR	
8	30.14M	COR	2.10	32	7.74M	PTR		56	4.46M	PTR		80	3.15M	PTR	
9	26.81M	COR	2.06	33	7.50M	PTR		57	4.38M	PTR		81	3.12M	PTR	
10	24.64M	PTR		34	7.29M	PTR		58	4.31M	PTR		82	3.08M	PTR	
11	22.42M	PTR		35	7.08M	PTR		59	4.23M	PTR		83	3.03M	PTR	
12	20.55M	PTR		36	6.89M	PTR		60	4.17M	PTR		84	3.01M	PTR	-
13	18.97M	PTR		37	6.70M	PTR	-	61	4.10M	PTR	-	85	2.98M	PTR	
14	17.61M	PTR		38	6.53M	PTR		62	4.03M	PTR		86	2.94M	PTR	
15	16.44M	PTR		39	6.36M	PTR		63	3.97M	PTR		87	2.91M	PTR	
16	15.42M	PTR		40	6.20M	PTR		64	3.91M	PTR		88	2.88M	PTR	
17	14.52M	PTR	-	41	6.06M	PTR		65	3.85M	PTR		89	2.85M	PTR	
18	13.71M	PTR		42	5.91M	PTR		66	3.80M	PTR		90	2.82M	PTR	
19	12.99M	PTR		43	5.78M	PTR		67	3.74M	PTR		91	2.79M	PTR	
20	12.35M	PTR		44	5.65M	PTR		68	3.69M	PTR		92	2.76M	PTR	
21	11.76M	PTR		45	5.53M	PTR		69	3.64M	PTR		93	2.73M	PTR	
22	11.23M	PTR		46	5.41M	PTR		70	3.59M	PTR		94	2.71M	PTR	
23	10.75M	PTR		47	5.29M	PTR		71	3.54M	PTR					
24	10.30M	PTR		48	5.18M	PTR		72	3.49M	PTR					

8.8 Summary

With this chapter we conclude the work of the thesis, by merging together the experience on the architecture model for the Tile*Pro*64 and the study on different memory allocation policies.

We implemented several versions of a parallel module for a video processing application, using the farm parallel pattern. Keeping the same pattern allowed us to test different implementations and compare them, as the parallel compiler of our programming framework would do.

For each implementation we estimated the service time with a variable number of workers. This allowed us to apply the architecture model to a real application (more complex w.r.t the benchmarks of Chapter 5) to:

- exemplify the selection of the architecture model parameters for a real parallel pattern implementation, in which we need to extract some values from the profiling of a sequential application, and we have supporting processes that behave differently w.r.t the set of workers;
- verify the approximation error of the architecture model with a more complex behavior

A low approximation error is fundamental in our approach, as we use the model to derive the service time function $T_s^{(n)}$ of the module, to let the compiler

- select the best implementation for each parallelism degree, and
- predict the service time of the module to select a parallelism degree sufficient to remove the bottleneck in the graph of modules composing the application.

We showed that, by using the model, the compiler usually select the correct implementation (i.e. the faster); there were, however, some cases in which the service times of the different implementations were very similar, and the compiler ended selecting a different, but still good, implementation that resulted in performance loss below the 1.5% w.r.t the best. Even our prediction on the service times of the module is acceptable, with errors within the 2.5% and thus extremely valid to let the compiler select the proper parallelism degree to remove the bottleneck in the graph of modules.

Given the good approximations obtained, we were able to use the model to estimate the behavior of the implementations on a multi-chip Tile*Pro*64 configuration. This allowed us to evaluate the possible performance impact of the different implementations in a more complex environment, in which remote memory latencies are increased because of inter-chip communications. This scenario makes indeed a lot more sense for the implementations provided, because of a more interesting behavior of the message-passing based parallelization, which is able to outperform the others when emitter and collector are not bottlenecks.

Chapter 9

Conclusions

With this thesis we studied performance models and optimizations for multi-core architectures, as an initial an effort inside the long-term project of ASSISTANT.

In particular, we focused on the performance prediction of parallel patterns, extending the approach already presented in [125] to consider multiple implementations for each module. To achieve this result, we developed an architecture model for a specific commercial architecture: the Tilera Tile*Pro*64. This architecture constitute an interesting insight of the future evolution of chip multiprocessor, given its 64 cores and the use of innovative solutions to handle the interconnection network and cache coherence problems. The model, based on queueing networks, can be considered an extension of the original approach of Bhandarkar [36] to evaluate the effects of contentions on the memory subsystem of a shared-memory parallel machine. By applying this model we are able to estimate the average *Response Time* of the memory, a fundamental parameter required to estimate the performance of the parallel code. While maintaining the original ideas, our model strongly differ from the ones previously presented in literature because of the intrinsic characteristics of the Tile*Pro*64 architecture. This model (that is able to estimate the response time with an average error below $\sim 10\%$, constitute, in our opinion, a very important research outcome because, as of today, no other models are able to estimate this parameter, with this accuracy, on commercially available architectures.

Following a different path, we also studied specific multicore-oriented optimizations for parallel patterns. We still focused on the impact of the memory subsystem, and in particular studied efficient uses of:

- **Multiple Memory Controllers**
- **Hardware Cache Coherence Mechanisms**

For the first aspect, we deeply analyzed the problem of having multiple memory interfaces on the same parallel machine. This represent an old problem, as already

present on off-chip multiprocessors. However, in this area multi-cores differs from both SMP and NUMA architecture; how to exploit them at best has not yet approached systematically by the research world. Currently, the limited number of works available is focusing on the problem from the operating-system point of view, i.e. how to *automatically* allocate and/or move virtual memory pages to optimize the global performance of the machine, in terms of completion times and/or power consumption. We approached the problem from the parallel-pattern point of view, i.e. how to structure a parallel module to better exploit the multiple memory controllers available. Our results shows that, in general, handling the multi-core as a NUMA machine help improving the performances, especially with a limited number of workers, because we are able to reduce the average memory request latency.

For the second aspect, given the increased complexity of multi-core architectures, an efficient use of the hardware cache coherency mechanisms allowed us to obtain significant performance improvements for both a stream- and a data-pattern on the Tile*Pro*64 processor. We also demonstrated that, by using a structured parallel programming approach, we are able to completely *disable* the hardware mechanisms and implement efficient software-based protocols in a transparent way to the programmer. Moreover, these protocols perform as good as or even better than the automatic approaches.

Finally, in the last part, we merged the two paths by demonstrating the use of the performance model and the optimizations introduced in the thesis to produce different implementations of a farm pattern, and compare them analytically to:

- estimate the service time of the module with an approximation $\leq 2.5\%$;
- select, for each parallelism degree, the best implementation.

The path towards ASSISTANT

As previously stated, this thesis has to be considered a small, yet very important, tile in our long-term project. In particular, the use of performance models is pervasive in our approach, as they are used both at compile and at run time: first to select the best implementation (using the available data), and then to drive the adaptation policies[128] of the dynamic run-time system.

Yet the road is still long: with this thesis we demonstrated the possibility of producing architecture-oriented performance models; however, each target architecture will need a new, especially made, model to capture the specific characteristics of the processor.

Even from the optimization point of view, many other aspects may be addressed in the future. In particular we believe that a further study of cache coherence mechanisms is required, because of the promisingly results obtained in this thesis.

Bibliography

[1] S. V. Adve, V. S. Adve, M. D. Hill, and M. K. Vernon. Comparison of hardware and software cache coherence schemes. *SIGARCH Comput. Archit. News*, 19:298–308, April 1991.

[2] V. S. Adve and M. K. Vernon. Parallel program performance prediction using deterministic task graph analysis. *ACM Trans. Comput. Syst.*, 22(1):94–136, Feb. 2004.

[3] M. Ajmone Marsan, G. Balbo, G. Conte, and F. Gregoretti. Modeling bus contention and memory interference in a multiprocessor system. *Computers, IEEE Transactions on*, C-32(1):60–72, 1983.

[4] M. Aldinucci. Automatic program transformation: The Meta tool for skeleton-based languages. In S. Gorlatch and C. Lengauer, editors, *Constructive Methods for Parallel Programming*, Advances in Computation: Theory and Practice, chapter 5, pages 59–78. Nova Science Publishers, NY, USA, 2002.

[5] M. Aldinucci, S. Campa, M. Danelutto, P. Kilpatrick, and M. Torquati. Targeting distributed systems in fastflow. In I. Caragiannis, M. Alexander, R. Badia, M. Cannataro, A. Costan, M. Danelutto, F. Desprez, B. Krammer, J. Sahuquillo, S. Scott, and J. Weidendorfer, editors, *Euro-Par 2012: Parallel Processing Workshops*, volume 7640 of *Lecture Notes in Computer Science*, pages 47–56. Springer Berlin Heidelberg, 2013.

[6] M. Aldinucci, M. Coppola, and M. Danelutto. Rewriting skeleton programs: How to evaluate the data-parallel stream-parallel tradeoff. In S. Gorlatch, editor, *Proc of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming*, pages 44–58. Fakultät für mathematik und informatik, Uni. Passau, Germany, May 1998.

[7] M. Aldinucci, M. Coppola, M. Vanneschi, C. Zoccolo, and M. Danelutto. Assist as a research framework for high-performance grid programming environments. In J. Cunha and O. Rana, editors, *Grid Computing: Software Environments and Tools*, pages 230–256. Springer London, 2006.

[8] M. Aldinucci, M. Danelutto, and P. Kilpatrick. Skeletons for multi/many-core systems. In *Parallel Computing: From Multicores and GPU's to Petascale (Proc. of PARCO 2009, Lyon, France)*, 2010.

[9] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati. An efficient unbounded lock-free queue for multi-core systems. In *Proc. of 18th Intl. Euro-Par 2012 Parallel Processing*, LNCS, Rhodes Island, Greece, Aug. 2012. Springer. To appear.

[10] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in java. *Future Generation Computer Systems*, 19(5):611 – 626, 2003. ¡ce:title¿Tools for Program Development and Analysis. Best papers from two Technical Sessions, at ICCS2001, San Francisco, CA, USA, and ICCS2002, Amsterdam, The Netherlands¡/ce:title¿.

[11] M. Aldinucci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Towards parallel programming by transformation: The fan skeleton framework. *Parallel Algorithms and Applications*, 16(2–3):87–122, Mar. 2001.

[12] M. Aldinucci, M. Meneghin, and M. Torquati. Efficient smith-waterman on multi-core with fastflow. In M. Danelutto, T. Gross, and J. Bourgeois, editors, *Proc. of Intl. Euromicro PDP 2010: Parallel Distributed and network-based Processing*, Pisa, Italy, Feb. 2010. IEEE.

[13] J. Ansel. *Autotuning Programs with Algorithmic Choice*. Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, February 2014.

[14] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. *SIGPLAN Not.*, 44(6):38–49, June 2009.

[15] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52:56–67, October 2009.

[16] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 319–330, New York, NY, USA, 2010. ACM.

[17] E. Ayguad, R. Badia, F. Igual, J. Labarta, R. Mayo, and E. Quintana-Ort. An extension of the starss programming model for platforms with multiple gpus. In H. Sips, D. Epema, and H.-X. Lin, editors, *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 851–862. Springer Berlin Heidelberg, 2009.

[18] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3l: A structured high-level parallel language, and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.

[19] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. Skie: A heterogeneous environment for {HPC} applications. *Parallel Computing*, 25(1314):1827 – 1852, 1999.

[20] F. Baiardi, L. Ricci, and M. Vanneschi. Static checking of interprocess communication in ecsp. *SIGPLAN Not.*, 19(6):290–299, June 1984.

[21] D. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. Simon, V. VenkataKrishnan, and S. Weeratunga. The nas parallel benchmarks summary and preliminary results. In *Supercomputing, 1991. Supercomputing '91. Proceedings of the 1991 ACM/IEEE Conference on*, pages 158–165, 1991.

[22] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu, and D. Zappala. Improving simulation for network research. Technical Report 99-702b, University of Southern California, March 1999. revised September 1999, to appear in IEEE Computer.

[23] S. Balsamo, V. D. N. Person, and P. Inverardi. A review on queueing network models with finite capacity queues for software architectures performance prediction. *Performance Evaluation*, 51(24):269 – 288, 2003. ¡ce:title¿Queueing Networks with Blocking¡/ce:title¿.

[24] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In *Hardware/Software Codesign, 2002. CODES 2002. Proceedings of the Tenth International Symposium on*, pages 73–78, 2002.

[25] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *J. ACM*, 22(2):248–260, Apr. 1975.

[26] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 - processor: A 64-core soc with mesh interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88 –598, feb. 2008.

[27] Y. Ben-Itzhak, E. Zahavi, I. Cidon, and A. Kolodny. Hnocs: Modular open-source simulator for heterogeneous nocs. In *Embedded Computer Systems (SAMOS), 2012 International Conference on*, pages 51–57, 2012.

[28] J. Berthold, M. Dieterle, R. Loogen, and S. Priebe. Hierarchical master-worker skeletons. In P. Hudak and D. Warren, editors, *Practical Aspects of Declarative Languages*, volume 4902 of *Lecture Notes in Computer Science*, pages 248–264. Springer Berlin Heidelberg, 2008.

[29] M. Bertoli, G. Casale, and G. Serazzi. Jmt: performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.*, 36(4):10–15, 2009.

[30] C. Bertolli, D. Buono, S. Lametti, G. Mencagli, M. Meneghin, A. Pascucci, and M. Vanneschi. A programming model for high-performance adaptive applications on pervasive mobile grids. In *Proceeding of the 21st IASTED International Conference on Parallel and Distributed Computing and Systems*, 2009.

[31] C. Bertolli, D. Buono, G. Mencagli, M. Mordacchini, F. M. Nardini, M. Torquati, and M. Vanneschi. Resource discovery support for time-critical adaptive applications. In *The 6th International Wireless Communications and Mobile Computing Conference. Workshop on Emergency Management: Communication and Computing Platforms*, 2010.

[32] C. Bertolli, D. Buono, G. Mencagli, and M. Vanneschi. Expressing adaptivity and context-awareness in the assistant programming model. In *Proceedings of the Third International ICST Conference on Autonomic Computing and Communication Systems*, 2009.

[33] C. Bertolli, D. Buono, G. Mencagli, and M. Vanneschi. Expressing adaptivity and context-awareness in the assistant programming model. In *Proceedings of the Third International ICST Conference on Autonomic Computing and Communication Systems*, 2009.

[34] C. Bertolli, D. Buono, G. Mencagli, and M. Vanneschi. An approach to mobile grid platforms for the development and support of complex ubiquitous applications. In *Handbook of Research on Mobility and Computing: Evolving Technologies and Ubiquitous Impacts*, 2011.

[35] C. Bertolli and M. Vanneschi. Fault tolerance for data parallel programs. *Concurrency and Computation: Practice and Experience*, 23(6):595–632, 2011.

[36] D. Bhandarkar. Analysis of memory interference in multiprocessors. *Computers, IEEE Transactions on*, C-24(9):897–908, 1975.

[37] L. Bhuyan, Q. Yang, and D. Agrawal. Performance of multiprocessor interconnection networks. *Computer*, 22(2):25–37, 1989.

[38] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[39] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[40] F. Blagojevic, P. Hargrove, C. Iancu, and K. Yelick. Hybrid pgas runtime support for multicore nodes. In *Fourth Conference on Partitioned Global Address Space Programming Model (PGAS10), Oct 2010*, 2010.

[41] D. Buono, M. Danelutto, T. De Matteis, G. Mencagli, and M. Torquati. A lightweight run-time support for fast dense linear algebra on multi-core. In *Proceedings of the 12th IASTED International Conference on Parallel Distributed Computing and Networks*, 2014.

[42] D. Buono, M. Danelutto, and S. Lametti. Map, reduce and mapreduce, the skeleton way. *Procedia Computer Science*, 1(1):2095 – 2103, 2010.

[43] D. Buono, M. Danelutto, S. Lametti, and M. Torquati. Parallel patterns for general purpose many-core. In *Proceeding of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013.

[44] D. Buono, M. Danelutto, S. Lametti, and M. Torquati. Parallel patterns for general purpose many-core. In *Parallel, Distributed and Network-Based Processing (PDP), 2013 21st Euromicro International Conference on*, pages 131–139, 2013.

[45] D. Buono, T. De Matteis, and G. Mencagli. A high-throughput and low-latency parallelization of window-based stream joins on multicores. In *Parallel and Distributed Processing with Applications (ISPA), 2014 IEEE 12th International Symposium on*, 2014. To Appear.

[46] D. Buono, T. De Matteis, G. Mencagli, and M. Vanneschi. Towards a methodology for parallel data stream processing: application to parallel stream join. Technical report, University of Pisa, 2013.

[47] D. Buono, T. D. Matteis, G. Mencagli, and M. Vanneschi. Optimizing message-passing on multicore architectures using hardware multi-threading. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 262–270, Feb 2014.

[48] D. Buono and G. Mencagli. Run-time Mechanisms for Fine-Grained Parallelism on Network Processors: the TILEPro64 Experience. In *High Performance Computing and Simulation (HPCS), 2014 International Conference on*, 2014. To Appear.

[49] D. Buono, G. Mencagli, A. Pascucci, and M. Vanneschi. Performance analysis and structured parallelisation of the spacetime adaptive processing computational kernel on multi-core architectures. *International Journal of Parallel, Emergent and Distributed Systems*, 0(0):1–39, 0.

[50] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[51] M. Butler, L. Barnes, D. Sarma, and B. Gelinas. Bulldozer: An approach to multithreaded compute performance. *Micro, IEEE*, 31(2):6–15, March 2011.

[52] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks. Helix: Automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 84–93, New York, NY, USA, 2012. ACM.

[53] B. Chamberlain, D. Callahan, and H. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21:291–312, August 2007.

[54] M. K. Chan and L. Yang. Comparative analysis of openmp and mpi on multi-core architecture. In *Proceedings of the 44th Annual Simulation Symposium*, ANSS '11, pages 18–25, San Diego, CA, USA, 2011. Society for Computer Simulation International.

[55] X. Chang. Network simulations with OPNET. In *Simulation Conference Proceedings, 1999 Winter*, volume 1, pages 307–314 vol.1, 1999.

[56] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[57] H. Cheong and A. Veidenbaum. Compiler-directed cache management in multiprocessors. *Computer*, 23(6):39–47, 1990.

[58] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 155–166, Washington, DC, USA, 2011. IEEE Computer Society.

[59] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Evaluation and improvements of programming models for the intel scc many-core processor. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 525–532, 2011.

[60] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An evaluation of global address space languages: co-array fortran and unified parallel c. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 36–47, New York, NY, USA, 2005. ACM.

[61] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, 2004.

[62] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the amd opteron processor. *IEEE Micro*, 30:16–29, 2010.

[63] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: towards a realistic model of parallel computation. *SIGPLAN Not.*, 28:1–12, July 1993.

[64] D. Culler, J. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1st edition, 1998. The Morgan Kaufmann Series in Computer Architecture and Design.

[65] T. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. Singh. From opencl to high-performance hardware on fpgas. In *Field Programmable Logic and Applications (FPL), 2012 22nd International Conference on*, pages 531–534, Aug 2012.

[66] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[67] M. Danelutto. Efficient support for skeletons on workstation clusters. *Parallel Processing Letters*, 11(01):41–56, 2001.

[68] M. Danelutto and M. Stigliani. Skelib: Parallel programming with skeletons in c. In A. Bode, T. Ludwig, W. Karl, and R. Wismller, editors, *Euro-Par 2000 Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 1175–1184. Springer Berlin Heidelberg, 2000.

[69] J. Darlington, Y.-k. Guo, H. W. To, and J. Yang. Parallel skeletons for structured composition. *SIGPLAN Not.*, 30(8):19–28, Aug. 1995.

[70] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic management: A holistic approach to memory placement on numa systems. *SIGARCH Comput. Archit. News*, 41(1):381–394, Mar. 2013.

[71] T. De Matteis, F. Luporini, G. Mencagli, and M. Vanneschi. Evaluation of architectural supports for fine-grained synchronization mechanisms. In *Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Networks*, Innsbruck, Austria, 2013.

[72] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[73] M. Dubois and F. A. Briggs. Effects of cache coherency in multiprocessors. *Computers, IEEE Transactions on*, C-31(11):1083–1099, 1982.

[74] J. Dunigan, T.H., J. Vetter, I. White, J.B., and P. Worley. Performance evaluation of the cray x1 distributed shared-memory architecture. *Micro, IEEE*, 25(1):30–40, 2005.

[75] J. Dunigan, T.H., J. Vetter, and P. Worley. Performance evaluation of the sgi altix 3700. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 231–240, 2005.

[76] A. DURAN, E. AYGUAD, R. M. BADIA, J. LABARTA, L. MARTINELL, X. MARTORELL, and J. PLANAS. Ompss: A proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[77] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003.

[78] R. Fatoohi. Performance evaluation of the dual-core based sgi altix 4700. In *Computer Architecture and High Performance Computing, 2007. SBAC-PAD 2007. 19th International Symposium on*, pages 97–104, 2007.

[79] C. Fensch and M. Cintra. An os-based alternative to full hardware coherence on tiled cmps. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 355–366, 2008.

[80] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM.

[81] F. Franchetti, M. Püschel, Y. Voronenko, S. Chellappa, and J. M. F. Moura. Discrete Fourier transform on multicore. *IEEE Signal Processing Magazine, special issue on "Signal Processing on Platforms with Multiple Cores"*, 26(6):90–102, 2009.

[82] B. Franke and M. O'Boyle. A complete compiler approach to auto-parallelizing c programs for multi-dsp systems. *Parallel and Distributed Systems, IEEE Transactions on*, 16(3):234–245, 2005.

[83] H. Franke, J. Xenidis, C. Basso, B. M. Bass, S. S. Woodward, J. D. Brown, and C. L. Johnson. Introduction to the wire-speed processor and architecture. *IBM J. Res. Dev.*, 54:27–37, January 2010.

[84] M. Frigo and S. Johnson. Fftw: an adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384 vol.3, May 1998.

[85] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. *SIGPLAN Not.*, 33(5):212–223, May 1998.

[86] Golang.org Community. The Go Programming Language WebSite, 2014. `http://golang.org`.

[87] S. Gorlatch and S. Pelagatti. A transformational framework for skeletal programs: Overview and case study. In J. Rohlim, editor, *Proc. of Parallel and Distributed Processing. Workshops held in Conjunction with IPPS/SPDP'99*, volume 1586 of *LNCS*, pages 123–137, Berlin, 1999. Springer.

[88] S. Gorlatch, C. Wedler, and C. Lengauer. Optimization rules for programming with collective operations. In *Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings*, pages 492–499, 1999.

[89] Y. Guo, V. Vlassov, R. Ashok, R. Weiss, and C. A. Moritz. Synchronization coherence: A transparent hardware mechanism for cache coherence and fine-grained synchronization. *Journal of Parallel and Distributed Computing*, 68(2):165 – 181, 2008.

[90] A. Gupta and V. Kumar. The scalability of fft on parallel computers. *Parallel and Distributed Systems, IEEE Transactions on*, 4(8):922–932, 1993.

[91] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing cache architectures and coherency protocols on x86-64 multicore smp systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 413–422, New York, NY, USA, 2009. ACM.

[92] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[93] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109, 2010.

[94] R. Iakymchuk and P. Bientinesi. Modeling performance through memory-stalls. *SIGMETRICS Perform. Eval. Rev.*, 40(2):86–91, Oct. 2012.

[95] K. ichi Nomura, R. K. Kalia, A. Nakano, and P. Vashishta. A scalable parallel algorithm for large-scale reactive force-field molecular dynamics simulations. *Computer Physics Communications*, 178(2):73 – 87, 2008.

[96] E. Ipek, B. Supinski, M. Schulz, and S. McKee. An approach to performance prediction for parallel applications. In J. Cunha and P. Medeiros, editors, *Euro-Par 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 196–205. Springer Berlin Heidelberg, 2005.

[97] K. Irani and I. Onyuksel. A closed-form solution for the perfornance analysis of multiple-bus multiprocessor systems. *Computers, IEEE Transactions on*, C-33(11):1004–1012, 1984.

[98] J. Jeffers and J. Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Newnes, 2013.

[99] F. Jiao, N. Mahajan, J. Willcock, A. Chauhan, and A. Lumsdaine. Partial globalization of partitioned address spaces for zero-copy communication with shared memory. In *HiPC*, pages 1–10. IEEE, 2011.

[100] C. F. Joerg. *The cilk system for parallel multithreaded computing*. PhD thesis, Cambridge, MA, USA, 1995.

[101] P. J. Joseph, K. Vaswani, and M. J. Thazhuthaveetil. A predictive performance model for superscalar processors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 161–170, Washington, DC, USA, 2006. IEEE Computer Society.

[102] R. Kalla, B. Sinharoy, W. Starke, and M. Floyd. Power7: Ibm's next-generation server processor. *Micro, IEEE*, 30(2):7–15, 2010.

[103] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. Power7: Ibm's next-generation server processor. *IEEE Micro*, 30:7–15, 2010.

[104] J. Kelm, D. Johnson, W. Tuohy, S. S. Lumetta, and S. Patel. Cohesion: An adaptive hybrid memory model for accelerators. *Micro, IEEE*, 31(1):42–55, 2011.

[105] Khronos Group. OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems, 2014. `https://www.khronos.org/opencl`.

[106] M. Kistler, M. Perrone, and F. Petrini. Cell multiprocessor communication network: Built for speed. *Micro, IEEE*, 26(3):10–23, May 2006.

[107] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25:21–29, 2005.

[108] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 67(8):634 – 658, 2010. ¡ce:title¿Special Issue on Software and Performance¡/ce:title¿.

[109] G. Krawezik. Performance comparison of mpi and three openmp programming styles on shared memory multiprocessors. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '03, pages 118–127, New York, NY, USA, 2003. ACM.

[110] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal. Atac: a 1000-core cache-coherent processor with on-chip optical network. In *Proceedings of PACT '10*, New York, 2010.

[111] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[112] K. K. Leung. Load-dependent service queues with application to congestion control in broadband networks. *Performance Evaluation*, 50(1):27 – 40, 2002.

[113] M. Leyton and J. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 289–296, 2010.

[114] M. Leyton and J. M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *Proceedings of PDP '10*, 2010.

[115] J.-J. Li, C.-B. Kuan, T.-Y. Wu, and J. K. Lee. Enabling an opencl compiler for embedded multicore dsp systems. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 545–552, Sept 2012.

[116] F. Liu. *Analytically modeling the memory hierarchy performance of modern processor systems*. PhD thesis, North Carolina State University, 2011. AAI3463793.

[117] J. A. Lorenzo-Castillo, J. C. Pichel, F. F. Rivera, T. F. Pena, and J. C. Cabaleiro. A flexible and dynamic page migration infrastructure based on hardware counters. *The Journal of Supercomputing*, 65(2):930–948, 2013.

[118] T. Maeurer and D. Shippy. Introduction to the cell multiprocessor. *IBM journal of Research and Development*, 49(4):589–604, 2005.

[119] J. Marathe and F. Mueller. Hardware profile-guided automatic page placement for ccnuma systems. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 90–99, New York, NY, USA, 2006. ACM.

[120] J. Marathe, V. Thakkar, and F. Mueller. Feedback-directed page placement for ccnuma via hardware-generated memory traces. *Journal of Parallel and Distributed Computing*, 70(12):1204 – 1219, 2010.

[121] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. *SIGMETRICS Perform. Eval. Rev.*, 32(1):2–13, June 2004.

[122] M. Marsan and M. Gerla. Markov models for multiple bus multiprocessor systems. *Computers, IEEE Transactions on*, C-31(3):239–248, 1982.

[123] M. Marzolla. The `qnetworks` toolbox: A software package for queueing networks analysis. In K. Al-Begain, D. Fiems, and W. J. Knottenbelt, editors, *Analytical and Stochastic Modeling Techniques and Applications, 17th International Conference, ASMTA 2010, Cardiff, UK, Proceedings*, volume 6148 of *Lecture Notes in Computer Science*, pages 102–116. Springer, June14–16 2010.

[124] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, 23:44–55, March 2003.

[125] G. Mencagli. *A Control-Theoretic Methodology for Adaptive Structured Parallel Computations*. PhD thesis, University of Pisa, 2012.

[126] G. Mencagli and M. Vanneschi. Analysis of control-theoretic predictive strategies for the adaptation of distributed parallel computations. In *Proceedings of the First ACM Workshop on Optimization Techniques for Resources Management in Clouds*, ORMaCloud '13, pages 33–40, New York, NY, USA, 2013. ACM.

[127] G. Mencagli, M. Vanneschi, and E. Vespa. Control-theoretic adaptation strategies for autonomic reconfigurable parallel applications on cloud environments. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 11–18, 2013.

[128] G. Mencagli, M. Vanneschi, and E. Vespa. Reconfiguration stability of adaptive distributed parallel applications through a cooperative predictive control approach. In *Proceedings of the 19th International Conference on Parallel Processing*, Euro-Par'13, pages 329–340, Berlin, Heidelberg, 2013. Springer-Verlag.

[129] M. Meneghin. *An Optimization Theory for Structured Stencil-based Parallel Applications*. Ph.d. thesis, University of Pisa, Cambridge, MA, Feb 2010.

[130] M. Moadeli, A. Shahrabi, W. Vanderbauwhede, and P. Maji. An analytical performance model for the spidergon noc with virtual channels. *J. Syst. Archit.*, 56(1):16–26, Jan. 2010.

[131] D. Molka, D. Hackenberg, R. Schone, and M. Muller. Memory performance and cache coherency effects on an intel nehalem multiprocessor system. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 261–270, 2009.

[132] D. Molka, R. Schne, D. Hackenberg, and M. Mller. Memory performance and spec openmp scalability on quad-socket x86_64 systems. In Y. Xiang, A. Cuzzocrea, M. Hobbs, and W. Zhou, editors, *Algorithms and Architectures for Parallel Processing*, volume 7016 of *Lecture Notes in Computer Science*, pages 170–181. Springer Berlin Heidelberg, 2011.

[133] Y. Nishikawa, M. Koibuchi, M. Yoshimi, K. Miura, and H. Amano. An analytical network performance model for simd processor csx600 interconnects. *J. Syst. Archit.*, 57(1):146–159, Jan. 2011.

[134] R. Nishtala and K. A. Yelick. Optimizing collective communication on multicores. In *Proceedings of HotPar'09*, 2009.

[135] S. Oaks and H. Wong. *Java Threads*. O'Reilly, Sebastopol, CA, 3 edition, 2004.

[136] J. Odom, J. K. Hollingsworth, L. DeRose, K. Ekanadham, and S. Sbaraglia. Using dynamic tracing sampling to measure long running programs. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, SC '05, pages 59–, Washington, DC, USA, 2005. IEEE Computer Society.

[137] A. V. Oppenheim and R. W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[138] S. Owicki and A. Agarwal. Evaluating the performance of software cache coherence. *SIGARCH Comput. Archit. News*, 17:230–242, April 1989.

[139] S. Pelagatti. *A methodology for the development and the support of massively parallel programs*. PhD thesis, 1993.

[140] S. Pelagatti. Patterns and skeletons for parallel and distributed computing. chapter Task and data parallelism in P3L, pages 155–186. Springer-Verlag, London, UK, UK, 2003.

[141] J. Perez, R. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Cluster Computing, 2008 IEEE International Conference on*, pages 142–151, Sept 2008.

[142] C. Pheatt. Intel threading building blocks. *J. Comput. Small Coll.*, 23:298–298, April 2008.

[143] M. Poldner and H. Kuchen. On implementing the farm skeleton. *Parallel Processing Letters*, 18(1):117–131, 2008.

[144] J. Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[145] R. Riedlinger, R. Bhatia, L. Biro, B. Bowhill, E. Fetzer, P. Gronowski, and T. Grutkowski. A 32nm 3.1 billion transistor 12-wide-issue itanium processor for mission-critical servers. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 84–86, 2011.

[146] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A cycle accurate memory system simulator. *Computer Architecture Letters*, 10(1):16 –19, jan.-june 2011.

[147] S. Saini, D. Jespersen, D. Talcott, J. Djomehri, and T. Sandstrom. Performance comparison of sgi altix 4700 and sgi altix 3700 bx2. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, 2008.

[148] D. Sanchez, G. Michelogiannakis, and C. Kozyrakis. An analysis of on-chip interconnection networks for large-scale chip multiprocessors. *ACM Trans. Archit. Code Optim.*, 7:4:1–4:28, May 2010.

[149] F. Schmager, N. Cameron, and J. Noble. Gohotdraw: Evaluating the go programming language with design patterns. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 10:1–10:6, New York, NY, USA, 2010. ACM.

[150] Silicon Graphics International Corp. Performance and productivity breakthroughs with very large coherent shared memory: The sgi uv architecture. Technical report, 2012.

[151] D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. *J. Parallel Distrib. Comput.*, 28(1):65–83, July 1995.

[152] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, June 1998.

[153] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

[154] M. Steuwer, P. Kegel, and S. Gorlatch. Skelcl - a portable skeleton library for high-level gpu programming. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1176–1182, May 2011.

[155] J. Stone, D. Gohara, and G. Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, May 2010.

[156] C. Su, D. Li, D. S. Nikolopoulos, M. Grove, K. Cameron, and B. R. de Supinski. Critical path-based thread placement for numa systems. *SIGMETRICS Perform. Eval. Rev.*, 40(2):106–112, Oct. 2012.

[157] I. Tartalja and V. Milutinovic. Classifying software-based cache coherence solutions. *Software, IEEE*, 14(3):90–101, 1997.

[158] E. Tejedor, M. Farreras, D. Grove, R. M. Badia, G. Almasi, and J. Labarta. Clusterss: A task-based programming model for clusters. In *Proceedings of the 20th International Symposium on High Performance Distributed Computing*, HPDC '11, pages 267–268, New York, NY, USA, 2011. ACM.

[159] M. M. Tikir and J. K. Hollingsworth. Hardware monitors for dynamic page migration. *Journal of Parallel and Distributed Computing*, 68(9):1186 – 1200, 2008.

[160] Tilera Corp. *Tile Processor User Architecture Manual*.

[161] Tilera Corp. One-dimensional radix-2 fft on the tile processor. Technical report, Tilera Corp., 2011.

[162] Tilera Corp. TilePro Processor Family, 2012. `http://www.tilera.com/products/processors/TILEPro_Family`.

[163] Tilera Corp. TILE-Gx8072™ Processor White Paper. Technical report, Tilera Corp., 2013. Available on `www.tilera.com`.

[164] D. Towsley. Approximate models of multiple bus multiprocessor systems. *Computers, IEEE Transactions on*, C-35(3):220–228, 1986.

[165] T.-F. Tsuei and M. Vernon. A multiprocessor bus design model validated by system measurement. *Parallel and Distributed Systems, IEEE Transactions on*, 3(6):712–727, 1992.

[166] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *25 years of the international symposia on Computer architecture (selected papers)*, ISCA '98, pages 533–544, New York, NY, USA, 1998. ACM.

[167] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *Proceedings of HPCA '99*, 1999.

[168] T. Ungerer, B. Robič, and J. Šilc. A survey of processors with explicit multithreading. *ACM Comput. Surv.*, 35(1):29–63, Mar. 2003.

[169] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[170] L. G. Valiant. A bridging model for multi-core computing. In *Proceedings of the 16th Annual European Symposium on Algorithms*, ESA '08, pages 13–28, Berlin, Heidelberg, 2008. Springer-Verlag.

[171] M. Vanneschi. Heterogeneous hpc environments. In D. Pritchard and J. Reeve, editors, *Euro-Par98 Parallel Processing*, volume 1470 of *Lecture Notes in Computer Science*, pages 21–34. Springer Berlin Heidelberg, 1998.

[172] M. Vanneschi. The programming model of assist, an environment for parallel and distributed portable applications. *Parallel Comput.*, 28(12):1709–1732, 2002.

[173] M. Vanneschi. *Architettura degli Elaboratori*. Pisa University Press, 2013.

[174] A. Varga and R. Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, pages 60:1–60:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[175] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on cc-numa compute servers. *SIGOPS Oper. Syst. Rev.*, 30(5):279–289, Sept. 1996.

[176] M. K. Vernon and M. A. Holliday. Performance analysis of multiprocessor cache consistency protocols using generalized timed petri nets. *SIGMETRICS Perform. Eval. Rev.*, 14(1):9–17, May 1986.

[177] R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[178] D. T. Wang. *Modern Dram Memory Systems: Performance Analysis and Scheduling Algorithm*. PhD thesis, College Park, MD, USA, 2005. AAI3178628.

[179] P. D. Welch. A fixed-point fast fourier transform error analysis. *Audio and Electroacoustics, IEEE Transactions on*, 17(2):151–157, 1969.

[180] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, and A. Agarwal. On-chip interconnection architecture of the tile processor. *Micro, IEEE*, 27(5):15 –31, sept.-oct. 2007.

[181] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[182] B. Wilkinson and M. Allen. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[183] D. L. Willick and D. L. Eager. An analytic model of multistage interconnection networks. *SIGMETRICS Perform. Eval. Rev.*, 18(1):192–202, Apr. 1990.

[184] M. Woodacre, D. Robb, D. Roe, and K. Feind. The sgi altixtm 3000 global shared-memory architecture - white paper. Technical report, Silicon Graphics, Inc, 2003.

[185] X. Wu and V. Taylor. Performance characteristics of hybrid mpi/openmp implementations of nas parallel benchmarks sp and bt on large-scale multicore supercomputers. *SIGMETRICS Perform. Eval. Rev.*, 38(4):56–62, Mar. 2011.

[186] W. A. Wulf and C. G. Bell. C.mmp: A multi-mini-processor. In *Proceedings of the December 5-7, 1972, Fall Joint Computer Conference, Part II*, AFIPS '72 (Fall, part II), pages 765–777, New York, NY, USA, 1972. ACM.

[187] L. Yang, X. Ma, and F. Mueller. Cross-platform performance prediction of parallel applications using partial execution. In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 40–40, 2005.

[188] Q. Yang, L. Bhuyan, and B.-C. Liu. Analysis and comparison of cache coherence protocols for a packet-switched multiprocessor. *Computers, IEEE Transactions on*, 38(8):1143–1153, 1989.

[189] A. Zavanella. Skel-bsp: Performance portability for skeletal programming. In M. Bubak, H. Afsarmanesh, B. Hertzberger, and R. Williams, editors, *High Performance Computing and Networking*, volume 1823 of *Lecture Notes in Computer Science*, pages 290–299. Springer Berlin Heidelberg, 2000.

[190] J. Zhang. Characterizing the Scalability of Erlang VM on Many-core Processors. Master Thesis, 2011.