University of Pisa and Scuola Superiore Sant'Anna

Master Degree in Computer Science and Networking

Laurea Magistrale in Informatica e Networking

Master Thesis

Cost Models for Structured Parallel Programming on Shared Memory Architectures

Candidate

Alberto Bandettini

Supervisor

Prof. Marco Vanneschi

Academic Year 2010/2011

Ringraziamenti

Il ciclo universitario sta volgendo al termine e, come doveroso, è giusto ringraziare chi, nel corso di questi anni, mi è stato vicino e ha permesso tutto questo.

In primis quindi vorrei ringraziare Fabio, con cui ho affrontato buona parte delle difficoltà incontrate in questi ultimi due anni e, cosa non da poco, gli innumerevoli viaggi da Lucca. Detto chiaro: "Leporini, è fatta!"

Un altro sentito grazie va a Daniele Buono che ha sottratto tempo utile al suo lavoro per offrire il suo supporto (e il suo simulatore) allo studio effettuato per la tesi.

Voglio poi ringraziare il Prof. Vanneschi, che è stato un fondamentale punto di riferimento durante la mia vita universitaria.

E poi, ovviamente, tutta la mia famiglia che, da una parte, mi ha sempre incitato a non mollare e, dall'altra, ha sopperito a qualsiasi esigenza. Un enorme grazie va quindi a mio padre Guido e mia madre Emanuela, a Claudia, Matteo, Gemma, Roberto e Elena. Spero vivamente che la gioia del successo ripaghi anche solo in parte i sacrifici fatti.

Un caloroso grazie va anche a tutti gli amici di università che hanno riempito le mie giornate con battute e scherzi rendendo questo percorso a ostacoli molto, ma molto, meno arduo. Credetemi! Un grazie quindi (rullo di tamburi) ad Andrea Lottarini (in arte Gufetto) e al nostro mitico BitCreekPeer che non va scordato, al sindacalista Tiziano De Matteis ("bon per te Tizi"), a Andrea Bozzi (Ingegner Umberto Boszi che sta sempre in vacanza), ad Alessio Pascucci che ogni lunedì mattina ascolta con attenzione le mie prodezze calcistiche (Pascu sarà bene stampare anche questa settimana la classifica capocannonieri), a Luiggi, il Menca (forza Juve, siamo forti quest'anno), il sardo Vespa e Daniele Virgilio (anno 1.5 di MCSN) che, dato le sue doti di scrittura, di Virgilio ha solo il nome. E ancora, Andrea Farruggia con il suo

flebile tono di voce, Matteo Fulgeri (detto anche FullGay) e la sua compagna Federica Bertozzi che, a breve, si metteranno insieme perché fatti l'uno per l'altra anche se per ora non se ne sono resi conto. Anna, Paolo, Emilio (o Miglio), Nebbia, Manuel e chi più ne ha più ne metta. Anche gli altri del corso come Filippo, Davide, il gazzettino Angela, Emnet, o meglio i suoi capelli, che dopo MOD è in grado di leggere questa pagina, la Martinelli con le sue torte, il Giuliani, la fisica Francesca *and so on* senza scordarsi nessuno.

GRAZIE!

Contents

Chapter 1

Introduction

The *High Performance Computing* (HPC) field studies the hardware-software interaction and applications characterized by requirements for high processing bandwidth, low response time, high efficiency and scalability.

Currently, *multiprocessors* and *multi-cores* are an important evolution/revolution from the technological point of view. These architectures are very complex and heterogeneous systems with parallelism exploited at processes level. The trend in multi-cores architectures seems that the number of cores per chip is expected to double every two years. The idea is to substitute few complex and power-consuming CPUs with many smaller and simpler CPUs that can deliver better performance per watt. An important role is played by high bandwidth and low latency interconnection structures with limited degree (especially on-chip) while shared memory starts to be organized in hierarchies.

All this aspects have enormous implications from the software point of view. We point out that these architectures can be exploited efficiently provided that applications are able to do it. In spite of this relevant architectural change, the actual programming tools are at very low-level for a programmer without profound knowledge in the HPC field. Further, performance prediction and/or performance portability is missing or it is still in an initial phase. Summarizing, a wide gap still exists between shared memory architectures and parallel programming development tolls.

We advocate that a structured and methodological approach is able to reach this targets by mean of *structured parallelism programming* (or *skeleton* based parallel programming) in which a limited set of paradigms aims

to provide standard and effective rules for composing parallel computations in a machine independent manner. The programmers have to use paradigms to realize the parallel application. The freedom of the programmer is limited but if paradigms allow composition, parametrization and ad-hoc parallelism, they become very easy to use from the programmer point of view and very useful to optimize from the compiler point of view. In fact, having a fixed set of paradigms the compiler has to "reason" completely on them inserting optimizations that could be platform-dependent or choosing the best implementation for the underlying architecture. All this means performance improvement without direct intervention of programmers.

This important target is both application and architecture dependent and could be accomplished by a *performance cost model* in association with a simplified view of the concrete architecture, i.e. the so called *abstract architecture* [20]. Considering that, a parallel compiler must be supplied of

- an **abstract architecture**, that is a simplified view of the concrete architecture able to describe the essential performance properties and abstract from all the others that are useless. It aims to throw away details belonging to different concrete architectures and emphasizes all the most important and general ones. An abstract architecture for shared memory architectures could be the one in Figure 1.1 wherein there exist many processing nodes as processes and the interconnection structure is fully interconnected.

- a **cost model** associated to the abstract architecture. This cost model have to sum up all the features of the concrete architecture, the interprocess communication run-time support and the impact of the parallel application. Further, we strongly advocated that a cost model should be easy to use and conceptually simple to understand.

We remark that a complete and accurate cost model for these architectures is still missing and the aim of this thesis is just to give a contribution in this direction. We want to study how a detailed shared memory architecture-dependent cost model for parallel applications can be realized with particular care about the impact of the parallel application.

The aim is to use cost models in the compiler technology in order to statically performs optimizations for parallel applications in the same way

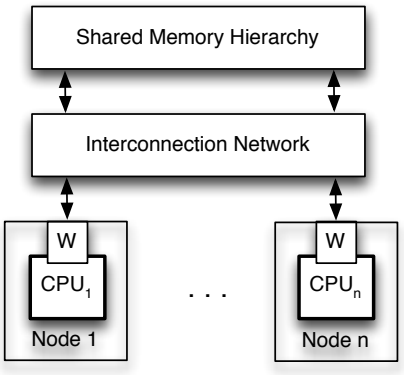


Figure 1.1: Simplified view of Shared Memory Architecture

that nowadays compilers do for sequential code. This should allow programmers to write in an easier way, i.e. using high-level and user-friendly tools, parallel applications that exploit the underlying architecture as well because compilers are able either to choose the right implementation or to use low-level libraries, that are very important for the performance point of view. Further, performance portability should be maintained among different concrete architectures. To our knowledge, there is no other work moving in this specific direction a part our main source of reference [20].

At processes level a parallel application can be viewed as a collection of cooperating processes via message passing. Formally, it is a graph wherein nodes are processes and arcs are communication channels among processes. This graph can be the result of a first compilation phase totally architecture independent and successively it can be easily mapped onto the abstract architecture for shared memory architecture because it has the same topology. All the outstanding concrete features are captured in two functions called T_{send} and T_{calc}. These functions are evaluated taking into account several characteristics of the concrete architecture, e.g. interconnection structure, processing node, memory access latency and so on. At this point, the parallel compiler has all the elements to introduce the architecture dependency according to the cost model. As already told, this way to operate allows optimizations or choices among various implementations in such a way performance predictability and/or portability can be achieved.

Anyway, the idea to sum up all the salient features of a concrete archi-

tecture in only two functions is, on one side, very powerful and easy to use but, on the other side, it is not a quite simple derivation.

In shared memory architectures various kind of resources are shared, e.g. memory modules and interconnection structures. The shared memory characteristic has, at the same time, pros and cons. On an hand it allows an easy way to design *run-time support* for interprocess communication, i.e. the implementation of *send* and *receive*, as an extension of the uniprocessor run-time support that takes into account important issues peculiar to shared memory architectures like *synchronization* or *cache coherence*. On the other hand, since all the processing nodes have to access the shared memory for loading data or to communicate, the memory becomes a source of performance degradation due to conflicts. So the effectiveness of the shared memory approach depends on the *latency* incurred on memory accesses as well as the *bandwidth* of information transfer that can be supported. We can consider conflicts on shared memory the major source of performance degradation in these architectures. Considering that, T_{send} and T_{calc} will be principally affected by this phenomenon so a cost model should describe this situation in a proper way in order to ensure at least performance prediction.

Formally, the impact of shared memory conflicts can be modelled as a *client-server* queuing system wherein clients C_i are processing nodes accessing the same macro-module while the server S is exactly that memory module, thus the under-load memory access latency is the *server response time* (conventionally called R_Q). Figure 1.2 shows this model that will be focus of interest in all the thesis.

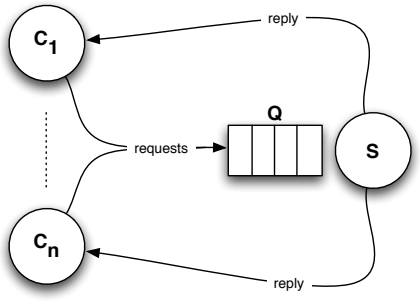


Figure 1.2: Client-Server System with Request-Reply behaviour.

In [20] the model is described through the following system of equations:

$$\begin{cases} T_{cl} = T_P + R_Q \\ R_Q = W_Q(T_s, T_A) + t_{a0} \\ \rho = \frac{T_S}{T_A} \\ T_A = \frac{T_{cl}}{p} \\ \rho < 1 \end{cases} \tag{1.1}$$

Each client C_i generates the next request only when the result of the previous one has been received. The behaviour of a client can be considered cyclic: local computational periods of average length T_P alternates to waiting ones (R_Q), leading to a certain client average inter-departure time T_{cl}. Once we know T_{cl} we can determine the server average inter-arrival time T_A as $\frac{T_{cl}}{p}$ applying the *Aggregate inter-arrival time* Theorem. Finally, the server response time R_Q is given by the average waiting time W_Q in the queue Q plus a constant known in advance that is the base latency t_{a0} of the server. Of course, W_Q depends on the type of queue placed in front of the server. The last expression points out that the system has a self-stabilizing behaviour (the utilization factor ρ of the server is less than one) so a steady-state solution exists. In this analytical approach we can find R_Q as resolution of a second degree equation in ρ.

In the following, the client-server model will be described in other formalisms, e.g. either as closed queuing network or as Continued Time Markov Chain (CTMC), and R_Q will be predicted through more resolution techniques, e.g. analytical and numerical. The reason is that we want to find a way to enhance the model for new behaviours and to improve its accuracy without increase the complexity of the resolution as much.

From this point of view, we know that Markov chains are a very powerful mathematical tool able to represent the behaviour of complex and concurrent systems as could be the Processors-Memory subsystem in shared memory architectures. Further, many numerical resolution techniques exist for moderately sized CTMC while iterative methods can be applied in case huge sizes are involved. Of course, Markov chains are difficult to build so we would

want to abstract from them and also from their resolution techniques.

For this purpose during the thesis, we will use a high level description language for Markov chains called *Performance Evaluation Process Algebra* (PEPA). It belongs to the *Stochastic Process Algebras* class and its usability comes out from the very formal interpretation of its expressions that is provided by an operational semantic. As we will see in Chapter 5, PEPA is a paradigm able to specify Markov chains that allows to express a complex system as composition of smaller components. These characteristics in addition to the high level approach, fit PEPA also as formalism to enhance and to solve the client-server model. To our knowledge, this is the first attempt to use PEPA for performance modelling in the HPC field.

We advocate that PEPA is *flexible* formalism for the client-server model able to reach *accuracy* in under-load memory access latency estimations and able to *accommodate* parallel application constraints.

For flexibility we mean a formalism able to adapt itself nimbly to even drastic architectural and/or application dependent changes. This ability is necessary in order to deal changes with no much effort and without increase a lot the resolution complexity of the model. A notable example could be the architectural passage from non-hierarchical shared memory to shared memory hierarchies that are very common in multi-cores architectures. For its relevance, this aspect will be treated in depth in a chapter.

Further, the accuracy aspect is very important for quantitative reasons. In order to be used, a performance cost model has to be precise. From this point of view, both analytical and numerical resolution techniques have been analysed and compared during the thesis. Of course, not always the more accurate solution is the best choice in terms of complexity so a good trade-off between this two contrasting requirements is needed.

Finally, we would want a formalism also able to taken into account the impact of the parallel application executed on the shared memory architecture. In other words, this means to satisfy application constraints. Notable examples could be an application composed by different processes or just processes exploiting a complex internal behaviour. We will treat this topics in depth.

Organization of the Thesis The thesis is organized in 8 chapters. The first one is just this Introduction that aims to focus on the context, the objective and the structure of this thesis.

Chapter 2 provides an overview of the main concepts about multiprocessors and multi-cores exploiting parallelism at processes level.

Chapter 3 summarizes the most important results of Queuing Theory that will be useful for future treatments. We recall that also the client-server model with request-reply behaviour reported in [20] is based on Queuing Theory.

Chapter 4 introduces two cost models for the Processors-Memory system: the former maps the system into a Closed Queuing Network while the latter is the client-server model already introduced. We will see pro and cons of both and their resolutions and we will propose a first variant of the second one taking into account a first application constraint: heterogeneous processes.

In order to enhance the model to take into account new architectural or application dependent aspects without increase the complexity, the PEPA formalism will be proposed in Chapter 5. Therefore, analysis and comparisons with other resolution techniques will be shown.

Chapter 6 examines the impact of parallel applications, i.e. applications composed either by different processes or with processes exploiting a complex internal behaviour. Also in this case, the theoretical contribution will be joined to experiments.

The shared memory hierarchy modelling and relative results will be treated in Chapter 7.

Finally, Chapter 8 draws the conclusions.

Chapter 2

Shared Memory Architectures

In this chapter we describe the main concepts about a class of parallel *Multiple Instruction Stream Multiple Data Stream* (MIMD) architectures: *multiprocessors* and *multi-cores* exploiting parallelism at processes level [20, 7, 17]. Obviously, we do not want to give a complete treatment of these architectures in this thesis, that can be found in [20, 7], but only the key concepts that will be used in chapters to come. So we will start summarizing the important topics for multiprocessors and successively we will extend them with particular care about multi-cores.

At first sight, a multiprocessor can be seen as a set of *processing nodes* that share one or more levels of memory hierarchy and are able to exchange firmware messages along an *interconnection structure*.

As we will see in this chapter, the processing nodes in a multiprocessor are *general purpose* CPUs possibly with a local memory and/or some I/O units while the interconnection structure is usually a trade off between performance and cost of the interconnection. The shared memory peculiarity means that any CPU is able to address any location of the physical memory. In other words, the result of the translation from logical addresses to physical ones can be any location of main memory. Moreover, lower levels in memory hierarchy can be shared.

The messages exchanged between processing nodes are used to implement shared memory accesses or explicit interprocessor communication, e.g. for process low-level scheduling like a decentralized process wake-up in *anonymous processors*. It is worthwhile to stress the fact that these messages are low level messages , so they must not be confused with messages at process

level.

The shared memory characteristic has, at the same time, pros and cons. On an hand it allows an easy way to design *run-time support* for interprocess communication, i.e. the implementation of *send* and *receive* if processes cooperate via message passing like in [20], because it is an extension of the uniprocessor run-time support, that takes into account important issues of *synchronization* or *cache coherence*. On the other hand, since all the processing nodes have to access the shared memory for loading data or to communicate, the memory becomes a source of performance degradation due to conflicts. So the effectiveness of the shared memory approach depends on the *latency* incurred on memory accesses as well as the *bandwidth* of information transfer that can be supported. Formally, this last aspect can be modelled as a *client-server* queuing system in which clients are processing nodes and servers are the memory modules, thus the memory access latency is the *server response time*. Anyway, we will see this model in very depth in the next chapter because it will be focus of interest in the rest of the thesis.

In the following of this chapter, we will deal with the structure and the properties of multiprocessors and relevant considerations will also be made for multi-cores.

Multi-cores, or *Chip MultiProcessor* (CMP), can be considered shared memory multiprocessors integrated on a single chip.

Therefore many results found for multiprocessor architectures are also valid for multi-cores, especially with a number of cores on the same chip relatively low. But the trend in shared memory architectures seems that the number of cores is expected to double every two years (Moore law applied to the number of cores on chip). In fact, the idea is to substitute few complex and power-consuming CPUs with many smaller and simpler CPUs that can deliver better performance per watt. It is worthwhile to point out that this last aspect is true provided that the software is able to exploit efficiently these architectures. In spite of this relevant architectural change, the actual programming tools are very low-level tools for a programmer without profound knowledge in this field. Further, performance prediction and/or performance portability is missing or it is still in an initial phase. As explained in [20], this targets could be accomplished by a *performance cost model*, that takes into account the features of different concrete architectures, in association

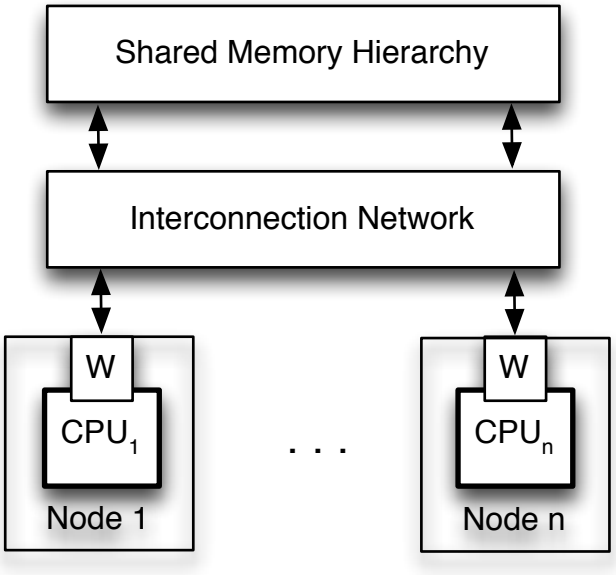


Figure 2.1: Simplified view of Shared Memory Architecture

with a simplified view of these architectures, the so called *abstract architecture*. Obviously, this is not a simple task but we will explain how it can also be achieved for complex architectures, like multiprocessors and multi-cores, using the structured and methodological approach utilized in [20]. However, we remark that a complete and precise cost model for these architectures is still missing and the aim of this thesis is to give a contribution in this direction with particular care about the impact of the parallel application. To achieve this goal is therefore necessary to investigate in depth important architectural factors, e.g. processing nodes, interconnection structures, shared memory hierarchy, cache coherence solutions and so on, and we are going to do this.

2.1 Processing Nodes

We focus on processing nodes that are composed by general purpose CPUs because in this way is possible to build multiprocessors or multi-cores on top of uniprocessor products exploiting all the advantages related to modularity. Successively, they themselves can be in turn building blocks for larger-scale

systems. This approach must however preserve the interoperability. For this purpose an *interface unit* W for each processing node is present. This unit has at least to be able to intercept all the memory requests and to transform them into proper firmware messages that will be sent either to the interconnection structure (*external* messages) or to some local units (*internal* messages) like a local memory (if present). Further, W has to be able to create proper firmware messages used for explicit communication between processing nodes.

It is important to notice that a potential re-utilization of uniprocessor architectures in greater contexts is not always free. In fact, some assembler and/or firmware mechanisms have to be already present in the uniprocessor design. Notable examples for shared memory architectures are synchronization mechanisms (requiring proper assembler instructions or annotations in the format of some assembler instruction) and cache management for the maintaining of coherent information among processing nodes. Anyway, this topics will be mentioned apart in the following.

A processing node in a shared memory architecture may have in general the schema visible in the Figure 2.2. As mentioned above, many features are common in both multiprocessors and multi-cores but slight differences may arise. We have:

- the CPU is a *pipeline* or *super-scalar* uniprocessor (with private data and instructions caches $L1$) exploding *Instruction Level Parallelism* (ILP), that is parallelism at firmware level. The CPU complexity can differ for various aspects that affect the performance of sequential code. In general, if it is required the maintenance of sequential performance, more complex CPUs are used. Otherwise, few large CPUs can be substituted with many simpler CPUs with a gain in efficiency and power consumption. Exclusively for multi-cores architectures, the CPU complexity could be influenced by the limited chip size because a trade-off between the features of each component on chip must be designed. It is worthwhile to note that if there are not hard constraints, hardware multi-threading, especially in the form of *Simultaneous Multi Threading* (SMT), is being used to exploit parallelism at firmware level

- the I/O *Communication Unit* (UC) is provided for explicit interpro-

cessor communication support. As we have mentioned above, though the majority of run-time support information are accessible in shared memory via memory instructions, there are some cases in which direct firmware messages between processing nodes are preferable. Notable examples are for processor synchronization and process low-level scheduling

- the interface unit W is directly connected to a local memory LM (if present) and some I/O units like UC. Moreover, dedicated links are also present toward the interconnection structure to allow information exchanges

- a local memory LM is used in general for caching information. This means to decrease the instruction service time as in uniprocessor architectures (local benefit) and, peculiarly of shared memory architectures, it reduces the shared memory conflicts as well as the interconnection structure congestion with a global performance improvement. The local memory may be a private memory of the processing node (for instance, it realizes the second or the third level of cache hierarchy) or, alternatively, LM may play a double role: it is integrating part of shared memory (so it can be addressed by all the other processing nodes) and, at the same time, it continues to operate as private memory support for the processing node. Exclusively for next generation multi-cores architectures, we can image that, looking at the Moore law applied to the number of cores, if LM is not private it will not be shared among all the cores but only among groups of cores due to performance degradations as a consequence of the slower memory access time

Looking at the explanation of a processing node in a shared memory architecture we can recognize that its structure is prevalently a uniprocessor architecture with firmware-assembler mechanisms to interoperate and coordinate with other processing nodes. So the structure of a processing node principally affects the performance of the sequential code but, in case a parallel application is executing on these architectures, the global performance depends by other factors like the impact of interconnection structures and/or the memory congestion. In the next sections we will treat this topics.

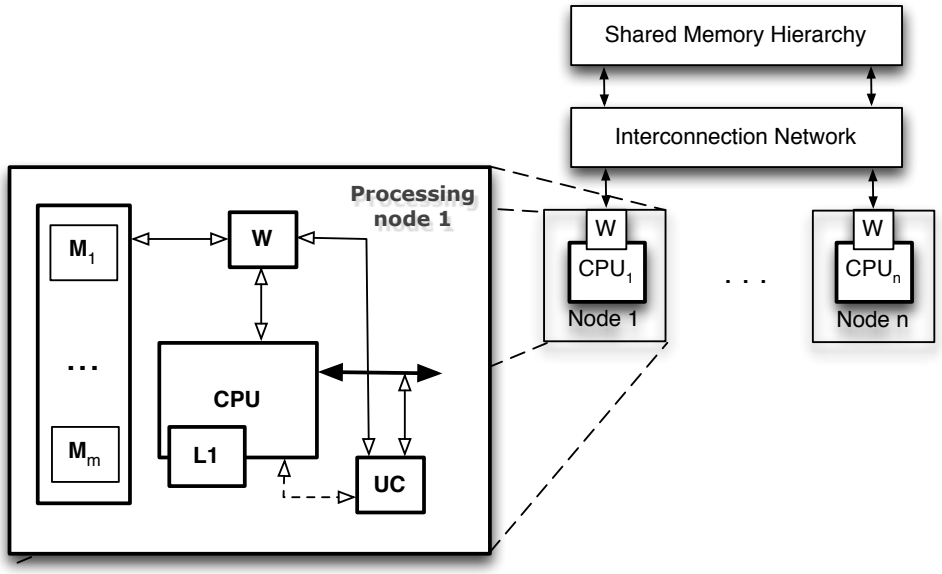


Figure 2.2: Processing Node in a Shared Memory Architecture

2.2 Interconnection Structures

The job of an interconnection structure in a MIMD parallel machine is to transfer firmware messages from any source node to any desired destination node in an efficient way that is, low latency and high bandwidth, that are features suitable for scalable highly parallel machines. This holds for both classes of MIMD architectures, i.e. shared memory multiprocessors and distributed memory multicomputers, but for the former class it is also important to do not fall into the pin count problem.

As we already know from the literature, many types of interconnection structure exist. In this context we do not want to list all of them with their features, that can be easily consulted in the literature, but we want to focus on same aspects that will be useful in future treatments. A detailed explanation of this topic can be found in [7].

In shared memory architectures, processing nodes communicate explicitly between them or with the memory modules across a sequence of links and switches. In the following, we will call all the entities that want to communicate through the interconnection structure, i.e. processing nodes and

memory modules, as nodes. As usually in networking domain, an interconnection structure (or network) can be formally viewed as a graph

$$N = (V, E)$$

where V is the set of nodes and switches and E is the set of links between them. The path from a source node to a destination node is called *route* and it is calculated by a *routing algorithm*. It is out of our scope to give complete treatment of routing strategies and algorithms so we only mention that routing can be *deterministic*, i.e. the path is determined solely by its source and destination, or *adaptive*, i.e. the choice of the path is influenced by dynamic events as traffic intensity along the way. Further, another important characteristic of a network is how information traverse the route (*switching strategy*). Basically, it may happen in *circuit switching*, i.e. the path between source and destination is established and reserved until it is necessary, or in *packet switching*. In the latter, the information are divided into packets individually routed from the source to the destination since each packet is carrying routing and sequencing information in addition to a portion of data. As we already know, this approach allows a better utilization of the network because resources are only occupied while a packet is traversing them so we will assume that interconnection structures that we are going to take into account will be packet switching networks.

From our point of view, it is important to understand that the above routing strategies can be directly accomplished by switches at firmware level. These units perform the so called *flow control* too. The flow control determines when a message can move along its route and it is absolutely necessary in case whenever two or more messages attempt to use the same network resource at the same time. For solving it, switches may adopt the classical *store-and-forward* technique or a more sophisticated strategy called *wormhole* flow control. In the latter, each packet is further subdivided in *flits*. Switches consider flits belonging to the same packet (that is still the unit of routing) as an input stream that must be forwarded in the same output port selected by the routing algorithm. Doing that, it is possible to achieve all the benefits due to a pipeline behaviour provided that the switches bandwidth per flit is high enough (and this holds because switches operate at firmware level). In this case, we can consider wormhole flow control as an

additional source of parallelism. Further, this technique has another important property: flits of the same packet are not entirely buffered before being forwarding so the buffering area is minimized. Owning to this property, the interconnection structures with this kind of flow control are very suitable for networks on chip because the smaller occupied area. Taking into account the increasing number of cores per chip these solutions are going to become very attractive for multi-cores architectures. In the following we will assume networks with wormhole flow control.

As mentioned above, an interconnection structure is a trade-off between cost of the interconnect and performance. Network latency and the bandwidth are critical parameters for measuring performance goodness while the number of bidirectional links and the occupied area should be considered for the cost of interconnect. Formally, we define the *base network latency* as the time needed to establish a communication through the network between a source and a destination *without contention*. In general it depends by many architectural characteristics, e.g average network distance, message length, routing, flow control strategy and so on, but it is not a function of the traffic. Instead, in case conflicts on the network are taken into account, we have the so called *under-load network latency*. For future aspects, it is important to evaluate in detail the base latency in a network with wormhole flow control. In the next subsection, we will see how this can be made.

2.2.1 Base Latency in Networks with Wormhole Flow Control

First of all, it is worthwhile to say that pipelined communications also occur in other firmware units (like memory interface units) that are involved in the path for achieving a destination, e.g. a memory module. Considering that, we can extend the wormhole behaviour to such that units and do not only consider the switches of the interconnection structure in the evaluation of the base latency.

As reported in [20], if we consider a firmware message of m words that travels d firmware units (as in the Figure 2.3) and assuming that

- flit size is equal to a word

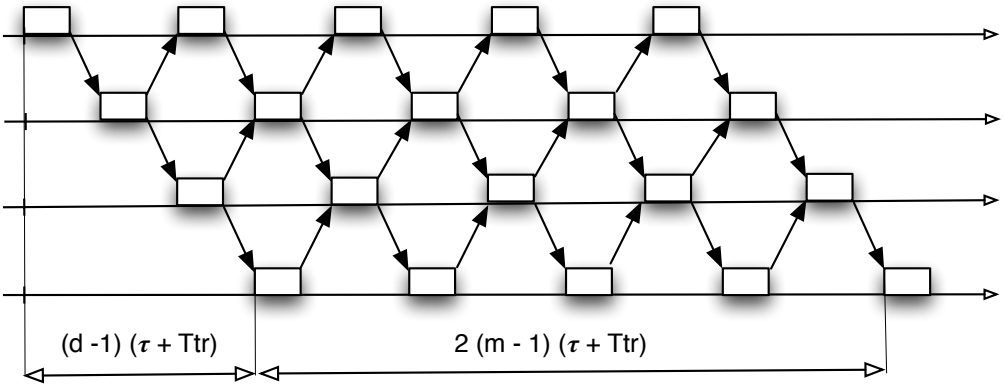


Figure 2.3: Base Latency in pipeline behaviour with level transaction firmware interfaces ($d = 4$ units and message length of $m = 5$ words)

- every unit has clock cycle τ
- every link has transmission latency T_{tr}
- level transaction firmware interfaces

we have that the base latency is

$$t_{a0} = (2m + d - 3)(\tau + T_{tr}) \tag{2.1}$$

2.2.2 Base Latency in Time-Slot Networks on chip

The above formula is in general a very good approximation but, in case the interconnection structure is completely on chip (like in multi-cores), further consideration should be taken into account. First of all, the transmission latency on chip is very negligible so $T_{tr} = 0$. Moreover, we have verified experimentally on a concrete architecture that firmware units do not wait for an acknowledgement before being sent the next flit since they are not more using level transaction protocols but specialized communication protocols, e.g. time slots based, are involved in order to reduce the latency (Figure 2.4). Therefore, the derivation of the base latency becomes

$$t_{a0} = (d + m - 2)\tau \tag{2.2}$$

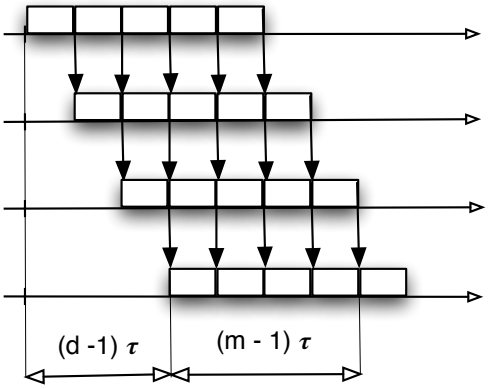


Figure 2.4: Base Latency in pipeline behaviour on chip ($T_{tr} = 0$) and time slot based communication firmware protocols ($d = 4$ units and message length of $m = 5$ words)

Another important fact came out from our study. The latency is further on reduced because units waiting for an information start to work immediately after the reception of the first part of the information, i.e. a word if the considered hierarchy is processor-first level cache or the first cache block if the hierarchy is first level cache-second level cache and so on, and no more after all the information.

2.2.3 Direct and Indirect Interconnection Structures

In the following of this section, we will study base latency and bandwidth as a function of the number n of nodes as in [20, 7]. As we can image, we would want low latency and high bandwidth on one side and we would not want to deal with the pin-count problem on the other side.

Since we are talking about interconnection structures for highly parallel machines, we should not take into consideration traditional buses that are no capable of simultaneous transfers (hence the bandwidth is $O(1)$) or fully connected crossbars (that are not physically realizable when the number n of nodes grows because the cost is $O(n^2)$). In spite of this, buses and crossbars are actually used. A notable example is the internal structure of switches that is usually a crossbar connecting all the input ports with all the output ports. Anyway, their use is only made in case the number of nodes is low. If

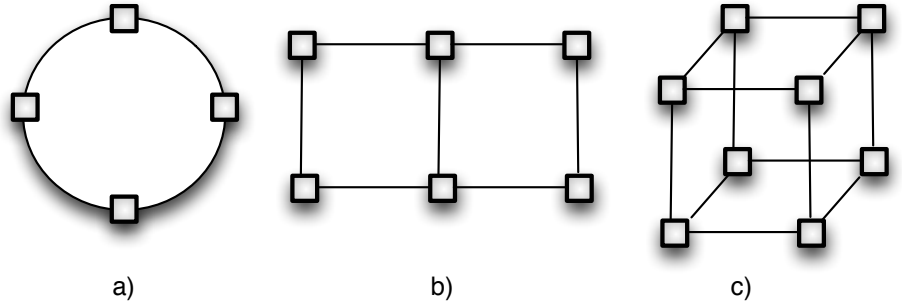


Figure 2.5: Most important Direct Networks with Limited Degree

an higher number of nodes is involved, other interconnection structures must be chosen.

In the so called *limited degree* networks, a node is directly connected to a small subset of other nodes or it is indirectly connected to every other node by an intermediate path of switches. These interconnection structures can be distinguished for their *topology* in *direct* or *indirect* networks.

In the former case, point-to-point dedicated links connect nodes in some fixed topology. Each node is connected to one and only one switch possibly through the interface unit W. Of course, communication between not adjacent nodes will travel intermediate nodes that will forward the information to the destination. Notable examples of direct networks are *rings* (2.5 a), *meshes* (2.5 b) and *cubes* (2.5 c).

In indirect networks, nodes are not directly connected as before but they are connected only with a subset of switches that, in turn, have a limited number of neighbours. i.e. other switches or nodes. In general, more than one switch is used in order to establish a communication between nodes. Notable examples are *trees*, *butterflies* and *fat trees*.

Briefly, we are going to summarize the most important features of some widely known interconnection structures. Rings have a base latency $O(n)$, meshes or two-dimensional cubes have $O(\sqrt{n})$ while butterflies, trees of fat trees $O(\log n)$. With respect to a tree, the so called fat tree has the channel capacity that doubles at each level from the leaves to the root in order to compensate the increasing congestion. It is worthwhile to say that all the cited interconnection structures connect nodes of the same type except the

butterfly that is principally used to allow communication between nodes of different types (for example n CPUs with n memory modules). Moreover, it can be used in a very elegant way to implement the so called *Generalized Fat Tree* that plays substantially two roles: it can be used as a fat tree and as a butterfly achieving an efficient solution in architectures where CPU-CPU and CPU-memory communications belong logically to different networks (like in UMA architectures).

Nowadays multiprocessors utilize more rings to communicate or, if the number of processing nodes is high, fat trees or generalized fat trees are used. In multi-cores architectures, the area of the chip is an hard constraint that limits many architectural choices as we have mentioned for the CPU complexity. Considering that, is not physically possible (until now) to fit complex interconnection structures on chip. Therefore, in case an high number of cores is involved, meshes are used because they have good scalability and easy realization on chip. A notable example of real multi-cores architecture using a mesh as interconnection structure is the Tilera TileGX. As we can see in Figure 2.6, the Tilera Tile64 is a 64 cores architecture specifically made for network processing in which every core has the instruction cache and data cache. Further, a second level of cache $L2$ realizes the private local memory. The next level in memory hierarchy is the shared memory level: there are four interfaces toward it at the borders of the chip. The mesh is used for explicit communication among cores but it also used for core-memory communication. In spite of this, it is important to notice that the mesh should only be used for the interconnection of the cores.

It is worthwhile to notice that, contrarily to many actual multi-cores, this architecture is not exploding an hierarchical shared memory organization. Anyway, shared memory hierarchies are widely coming out as a consequence of the integration of memories on chip, so nowadays the necessity to model this aspect should be treated. In the chapters to come we will deal with this.

2.3 Shared Memory

As we have said, a single physical address space across all the processing nodes in shared memory architectures is involved. This allows an easy way to design an efficient run-time support for interprocess communication be-

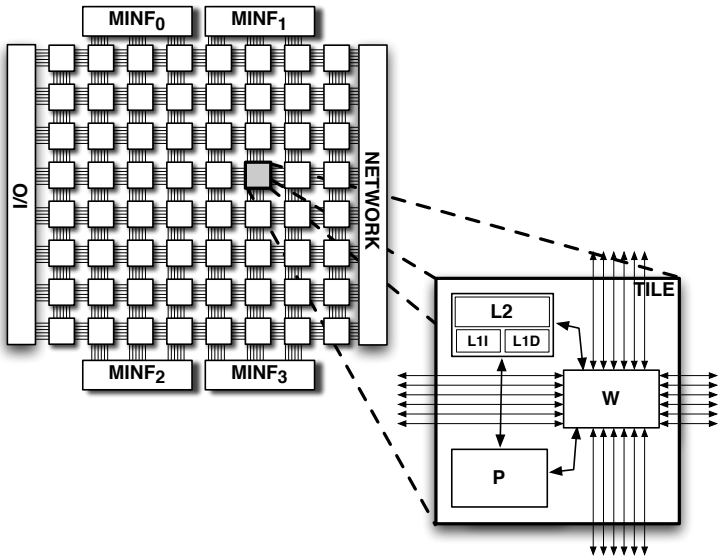


Figure 2.6: Tilera Tile64

cause it occurs implicitly as a result of conventional memory accesses instructions like happened in uniprocessor architectures. On the other hand, since many processing nodes are present, it may happen that more than one processing node wants to access the shared memory at the same time. Doing that, processing nodes cause congestion in the interconnection structure as well. Further, more hierarchical levels of shared memory could aggravate the congestion.

Firstly, assume that the congestion probability in the interconnection structure is very low. As soon as possible, we will see that this assumption is true in various conditions. At this point, we can assume that the major source of performance degradation is due to the queue in front of the shared memory. We already know that local memories may reduce conflicts on shared memory but, anyway, the design of the shared memory in a proper way is very important from the performance point of view. In particular, we need high bandwidth and a minimal contention on the memory. These goals can be achieved by mean of *modular memory with interleaved organization*. These memories are organized in macro modules with their own organization that can be interleaved or sequential. At this level, the interleaved organization has the principal effect to reduce the contention on memory modules.

Moreover, a single macro module can be realized either with an interleaved internal organization or with just one module with *long word*. Often, the number of the internal modules, or the number of words in a long word, coincides with the cache block size because it allows high bandwidth transfers of cache blocks.

2.3.1 UMA and NUMA Architectures

Another important point that is worth to explain is the shared memory organization in multiprocessor architectures. This characteristic is used to classify architectures on the base of the relative distance of the shared memory modules with respect to processing nodes. The accesses can be mainly performed in two styles:

1. in *uniform access memory* (UMA) memory accesses take the same time no matter which CPU requests them
2. in *non uniform access memory* (NUMA) some memory accesses are much faster than others depending on which CPU ask for which word or block

In the former, the memory modules are equidistant from the processing nodes. This means that the base latency to access them is the same independently both from the processing node and requested word. In spite of this symmetry, (private) local memories are used inside processing nodes in order to capitalize on the advantages about caching as mentioned above.

In the last memory organization, the symmetry about memory accesses is not more present. If we look at the typical schema of a processing node reported in Figure 2.2, we can consider the shared memory as the union of all the local memories of the processing nodes:

$$M = \bigcup_{i=1}^{n} LM_i$$

Hence LM is not more exclusively private of a processing node, but it can be accessed from the external ones. However, every processing node accesses own local memory in a very shorter time with respect to external ones that must travel the interconnection structure.

Thus, every processing node in shared memory multiprocessors has its own interface toward the memory so no conflicts are present in accessing the interface. This does not hold any more for multi-cores architectures because it is not physically realizable to put a memory interface in the same chip for every core. This constraint creates an ulterior source of performance degradation.

The distinction between UMA and NUMA shared memory organizations can be also effectuated for multi-cores. If the number of cores is low or there is a single interface, the architecture *seems* UMA, otherwise it should be considered NUMA. This is the case of the Tilera Tile64 (Figure 2.6) since it has four shared memory interfaces placed at the borders of the chip and all the cores access them though the mesh. Of course, different cores access the same memory module with different base latencies.

Until now we were assuming interconnection structures with low probability of conflicts. It is important to keep in mind that if this does not hold we should take into account the impact due to congestion on networks. Likely, we know that there exist networks (fat trees, generalized fat trees) that minimize the conflicts so we can assume the above property for at least multiprocessors. Instead, we can assume that it also holds for multi-cores because the performance degradation due to interconnection structures realized on chip is negligible with respect to the memory impact. The reasons should be the sophisticated techniques (as wormhole flow control and time slot based communication protocols) that have been used and the difference in frequency clock between the chip with the cores and the memory. This has further been verified experimentally on a real architecture from our research group and more information can be found in [15].

In the following, we will abstract from the interconnection structure and we concentrate only on shared memory performance degradation. However, it is worthwhile to stress that, even if interconnection structure congestion is negligible, the impact of the network latency must be taken into account from the performance point of view. In the next subsection we are going to see how the interconnection structure latency impact on the evaluation of the base memory access latency.

2.3.2 Base and Under-Load Memory Access Latency

The base latency to access a shared memory memory will be a fundamental parameter for our treatment. Assuming that the architecture is *unloaded*, i.e. any conflict is present, it can be defined as the time that a firmware unit, e.g. the second level cache, asking for a memory access, must wait before it receives (the first part of) the memory reply.

We can evaluate the base memory access latency using the approach introduced in 2.2.1. In fact, as we already told, if all the traversed firmware units operate in a pipeline behaviour we can easily estimate this value only knowing the number d of traversed units and the length of the memory request and reply firmware messages.

It is important to stress the fact that the base memory access time does not taken into all the time lost due to congestion on shared memory modules. Therefore, it cannot be a good evaluation of the time that a unit spent for waiting a reply in case conflict are present in the architecture. As we already defined for networks, if we add the impact of the congestion we have the so called *under-load memory access latency*. Following the approach presented in [20], in chapters to come we will see how this value can be estimated.

2.4 Synchronization and Cache Coherence

As we already mentioned, shared memory architectures must provide assembler-firmware mechanism for two critical aspects that are:

1. *synchronization*. Having many processing nodes and a physical shared memory space, shared information among processes can be accesses at the same time by different CPUs. If sequences of *indivisible operations* must be performed (like in interprocess communication run-time support), *locking* mechanisms are needed (in addition to disable interrupts like in uniprocessor)

2. *cache coherence*. As we have already told, in shared memory architectures caching is important for local and global performance improvement. However, in presence of private cache hierarchy per processing

node, shared information among processes must be maintained consistent because it may happen that in hierarchies of different processing nodes the value of the same data differs

If we consider processes that cooperate via message passing like in [20], synchronization and cache coherence mechanisms are used in the implementation of *send* and *receive* operations. As we will see in this section, the way to use these mechanisms can affect the performance so a proper design of the interprocess communication run-time support in shared memory architectures should be made.

Locking primitives are realized at assembler level with proper instructions (or annotations) and implement mutual exclusion of indivisible sequences among CPUs provided that *lock* and *unlock* themselves are atomic operations. This last property is directly implemented at firmware level by shared memory arbitration mechanisms, i.e. blocking the access to the memory module containing the locking semaphore. An efficient solution in terms of memory congestion and fairness, i.e. each CPU is guaranteed to access the lock section in finite time, is the so called *fair locking*. If the locking semaphore is realized in a proper way, from the performance point of view this technique does not introduce further degradation in addition to the cost of loading the cache block. Anyway, it is important to recall that in case a memory module is not accessible by a CPU, the CPU itself must wait. The increasing number of conflicts due to accesses in mutual exclusion is called *software lockout*. Although this phenomenon, its impact can be minimized realizing the run-time support in a proper way, i.e. minimizing the length of critical sections even though the number of critical sections grows. In the following, we assume a suitable interprocess communication run-time support so we will abstract from this problem. It is worthwhile to note that in multi-cores architectures the impact due to locking mechanisms implemented as shared memory accesses could be worse than in multiprocessors. This is due to architectural aspects: in multi-cores CPUs are on chip while shared memory modules are usually out of chip so a communication could be costly. In spite of this, actual multi-cores perform synchronization via atomic memory accesses.

The literature widely copes with the cache coherence problem. Exhaustive references are [7], [14]. First of all, we remark that information coherence is maintained in case of reading. Of course, writing is dangerous but only if the information is shared among processes. Briefly, we want to summarize some aspects concerning the adopted solutions. There are two main techniques for the maintenance of coherent information among CPUs:

1. *automatic cache coherence*. In the majority of the shared memory architectures, there exists firmware mechanisms that automatically guarantee the cache coherence

2. *algorithm-dependent cache coherence*. This approach is not characterized by automatic firmware mechanisms, but it is software-based. So the cache coherence is entirely managed by the programmer of the interprocess communication run-time support

In the former solution, the idea is to notify memory accesses (or better writings) to all the other CPUs. When a CPU is writing an information, the notification can be done by *update*, i.e. the modified information is sent to all CPUs, or *invalidation*, i.e. other copies become not valid as a consequence of an invalidation signalling. The latter may seem more complicated but it has a lower overhead because there is not need to communicate the entire modified information. Mainly, there are two implementation categories to these strategies:

1. the *snoopy-based* implementations use a centralization point at firmware level (snoopy bus) to notify modified information or invalidation messages. As we already know, buses are interconnection structures useful only when the number of nodes is low so this is not convenient if the number of CPU is high

2. *directory-based*. It is useful when the number of CPUs increases and more complex interconnection structures are used. The idea is to store in a memory support information about which cache contains which block. This approach reduces the overhead but it has a cost, i.e. congestion comes out in spite of the directory should be allocated in a fast memory

This last strategy could not make use of automatic firmware mechanisms. In that case we are talking about the so called algorithm-dependent cache coherence. As we mentioned above, software-based approaches should only be considered for the design of the interprocess communication run-time support. The idea is to capitalize on synchronization mechanisms to design an explicit management of cache coherence in the run-time support code. In theory, this approach does not introduce inefficiency per se but software lockout impact could be aggravated. Unlikely, nowadays is not possible to turn off automatic firmware mechanisms of cache coherence in the majority of shared memory architectures. A notable example of architecture in which is possible to disable automatic cache coherence is the Tilera Tile64 that we have already introduced. In the following, we will assume algorithm-dependent cache coherence and we abstract from it.

2.5 Cost Model and Abstract Architecture

We recall from the Introduction that the focus in this thesis is to study how a detailed shared memory architecture-dependent cost model of parallel applications can be realized. The aim is to use this cost model in the compiler technology in order to *statically* perform optimizations for parallel applications in the same way that nowadays compilers do for sequential code. This should allow programmers to write in an easier way, i.e. using high-level and user-friendly tools, parallel applications that exploit the underlying architecture as well because compiler are able either to choose the right implementation or to use low-level libraries that are very important from the performance point of view. Further, performance portability should be maintained among different architectures and other important properties should be guaranteed, e.g. adaptivity and dinamicity.

A good starting point to achieve the above targets is by mean of *structured parallel programming* (or *skeleton* based parallel programming) in which a limited set of skeleton describe the structure of particular styles of algorithm. The programmers have to use paradigms to realize the parallel application. The freedom of the programmer is limited but if paradigms allow composition, parametrization and ad-hoc parallelism (in addition to other important features as required in the definition) they become very easy to

use from the programmer point of view and very useful to optimize from the compiler point of view. In fact, having a fixed set of paradigms the compiler has to "reason" completely on them inserting optimizations that could be platform-dependent or choosing the best implementation for the underlying architecture. To achieve this goal, that is both application and architecture dependent, the compiler must be supplied of

- a simplified view of the concrete architecture, called **abstract architecture**, able to describe the essential performance properties and abstract from all the others that are useless. In fact, we have just seen that a shared memory architecture is a very complex system with many characteristics and mechanisms and to take into account all of them is hardly and it has no much sense. In fact, the abstract architecture aims to throw away details belonging to different concrete shared memory architectures and emphasizes all the most important and general ones. an abstract architecture for shared memory architectures could be the one in Figure 2.1 in which there exist many processing nodes as processes and the interconnection structure is fully interconnected

- a **cost model** associated to the abstract architecture. This cost model have to sum up all the features of the concrete architecture, the interprocess communication run-time support and the impact of the parallel application. Further, we strongly advocated that a cost model should be easy to use and conceptually simple to understand

As we can image, this is not a quite simple task for many different reasons but a wide contribute has been already proposed in [20]. To our knowledge, there is no other work moving in this specific direction.

We recall from [20] that at processes level a parallel application can be viewed as a collection of cooperating processes via message passing. Formally, it is a graph in which nodes are processes and arcs are communication channels among processes. This graph can be the result of a first compilation phase totally architecture independent and successively it can be easily mapped onto the abstract architecture for shared memory architecture because it has the same topology. Successively, all the outstanding concrete features are captured in two functions called T_{send} and T_{calc}. These functions

are evaluated taking into account several characteristics of the concrete architecture, e.g. interconnection structure, processing node, memory access time and so on. At this point, the parallel compiler has all the elements to introduce the architecture dependency. This allows the compiler to introduce optimizations or to choose between different implementations in case different concrete architectures are involved. It is worthwhile to note that this implies performance portability.

Anyway, the idea to sum up all the salient features of a concrete architecture in only two functions is, on an hand, very powerful and easy to use but, on the other hand, it is not a quite simple derivation. In the previous sections, we explained that we can consider conflicts on shared memory the major source of performance degradation in these architectures. Considering that, T_{send} and T_{calc} will be principally affected by this phenomenon so a model for describing this situation should be used. Formally, as we mentioned above, this aspect can be modelled as a *client-server* queuing system in which clients are processing nodes and servers are memory modules (including the interconnection structure impact). The shared memory access latency is the so called *server response time*. This model will be the focus of this thesis and we will explain it in depth in Chapter 4.2 but, firstly, it is necessary to recall some queuing theory concepts.

Chapter 3

Queueing Theory Concepts

We want to formalize a cost model basing on Queueing Theory concepts. Thus in this section we will refresh and summarize important results regarding both simple and intermediate queueing systems; the reader may consult [13, 5] for a deeper understanding of those concepts that here will be just reviewed.

3.1 Description and Characterization of a Queue

Description of Queues A queueing system models the behaviour of a server S where clients (often known as jobs or client requests) arrive and ask for a service. In general, clients have to spend some time in a queue Q waiting that S is ready to serve them. The scheme in Figure 3.1 is a

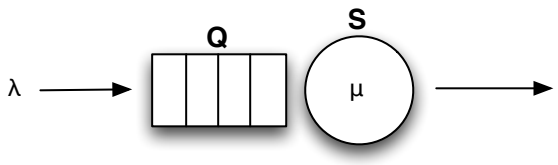


Figure 3.1: A queue

logical one, not necessarily corresponding to the real structure of the system we are modelling. For instance, Q could not physically exists or it could be even distributed among the clients. However, in some of these cases it turns out to be easier to study the whole system as a single logical queue. This

kind of approximation can drastically reduce the complexity of the analysis and makes it possible to obtain an approximate evaluation, which is however meaningful provided that the mathematical and stochastic assumptions are validated. We will use and explain this approach in the next sections.

Queue models are classified according to the following characteristics.

- The stochastic process A that describes the arrivals of clients. In particular, we are interested in the probability distribution of the random variable t_A extracted by A. t_A represents the *inter-arrival time*, that is the time interval between two consecutive arrivals of clients. Its mean value is denoted by T_A, the standard deviation by σ_A and the mean rate of inter-arrivals by $\lambda = \frac{1}{T_A}$.

- The stochastic process B that describes the service of S. B generates the random variable t_S that represents the *service time* of S, that is the time interval between the beginning of the executions on two consecutive requests. Its mean value is denoted by T_S, the standard deviation by σ_S and the mean rate of services by $\mu = \frac{1}{T_S}$.

- The *number of servers or channels* r of S, that is the parallelism degree of S. In the following, except for some specific cases, we will assume $r = 1$, that is a sequential server.

- The *queue size* d, that is the number of positions available in Q for storing the requests. Notice that in computer systems this size is necessarily fixed or limited. Unfortunately most of the results in Queueing Theory have been derived for infinite length queues. However, the results provided for infinite queues will sufficiently approximate the case of finite ones, under assumptions that we will discuss case by case.

- The *population* e of the system, which can be either infinite or finite.

- The *service discipline* x, that is the rule that specifies which of the queued requests will be served next. We will use the classical $FIFO$ discipline.

Basing upon these information, queues can be classified according to the standard Kendall's notation (see [13] for more details). For instance, we will

indicate with $M/M/1$ the queue with a single server where both input and service processes are Poisson ones.

Inter-departures Process The stochastic process C, that represents the departures from the system (inter-departure process), depends on the nature of the queue. For $A/B/1$ queues, being T_P the average inter-departure time, an evident result is that $T_P = max(T_A, T_S)$.

A first interesting property is the following (see [20] for a simple proof):

Theorem 1 ***Aggregate inter-arrival time.*** *If a queue Q has multiple sources (i.e. multiple arrival flows) each one with an average inter-departure time T_{p_i}, the total average inter-arrival time to Q is given by:*

$$T_A = \frac{1}{\sum_{i=1}^{N} \frac{1}{T_{p_i}}}$$

Characterization of Queues A first average measure of the *traffic intensity* at a queue is expressed through the *utilization factor* ρ.

$$\rho = \frac{\lambda}{\mu} = \frac{T_S}{T_A}$$

For our purposes an extremely important situation is given by $\rho < 1$. Under this situation the system *stabilizes*, therefore it becomes possible to determine the so called *steady-state* behaviour of the system.

Other metrics of interest to evaluate the performance of a queueing system are:

- the *mean number of requests in the system*, N_Q: the average number of client requests in the system including the one being served;
- the *waiting time distribution*: the time spent by a request in the waiting queue. We are practically interested in its mean value W_Q.
- the *response time distribution*: with respect to the waiting time distribution, it includes also the time spent in the service phase. We will denote its mean value as R_Q. Notice that $R_Q = W_Q + L_S$, where L_S is the average service latency .

A very general result that can be applied to different kind of scenarios (not just Queueing Theory) is the Little's theorem.

Theorem 2 ***Little's law.*** *Given a stable system ($\rho < 1$) where clients arrive with a rate λ and the mean number of clients in the system N_Q is finite, the average time spent by a client in the system R_Q is equal to*

$$R_Q = \frac{N_Q}{\lambda}$$

The reasoning behind this theorem is intuitive, while the proof is quite complicated. The interested reader may consult [13] for a deeper explanation.

3.2 Notably important Queues

The Queueing Theory is extensive and treats an incredible large number of special queues (that is, queues with a specific configuration $A/B/r/d/e/x$), some of which also particularly complicated. In order to keep limited the complexity of deriving the architecture cost model, we will be interested in a minimal (yet meaningful) subset of these queues. Therefore, in this section we illustrate the main results for only two peculiarly configurations: the $M/M/1$ and the $M/G/1$ queues.

3.2.1 The $M/M/1$ Queue

In a $M/M/1$ queue the arrivals occur according to a Poisson process with parameter λ. The services are exponentially distributed too, with rate μ. The memoryless property of the exponential distribution, besides being simple to model, is very important in our context because it allows us to approximate a lot of different meaningful scenarios. The service discipline is FIFO and it is assumed that the queue size is infinite. It can be shown that the average number of requests in the system is equal to

$$N_Q = \frac{\rho}{1-\rho}$$

Applying the Little's law we obtain:

$$W_Q = \frac{\rho}{\mu(1-\rho)}$$

$$R_Q = \frac{1}{\mu(1-\rho)}$$

It could be also proved that even if the queue has finite size k, the previous formulas still represent an acceptable result provided that the probability that a request gets stuck due to the full queue is an event with negligible probability.

3.2.2 The $M/G/1$ Queue

Although very common, the hypothesis on the exponential distribution of the service time could not be applicable in some concrete case of interests. For instance, there could be architectures in which the memory subsystem takes a *constant* amount of time to handle a processor request. In these cases we are interested in the deterministic distribution.

We introduce the $M/G/1$ queue, where the symbol G stands for *general* distribution. All assumptions and considerations made for the $M/M/1$ are still valid, except for the distribution of the services: indeed with an $M/G/1$ we are able to model any distribution of the service time. For this queue we get the following fundamental results (coming from the so called *Pollaczek-Khinchine formula*):

$$N_Q = \frac{\rho}{1-\rho}\,[1 - \frac{\rho}{2}(1-\mu^2\sigma_S^2)]$$

Applying the Little's law:

$$R_Q = \frac{1}{\mu(1-\rho)}\,[1 - \frac{\rho}{2}(1-\mu^2\sigma_S^2)]$$

A particular case of interest is the $M/D/1$ queue where the service time distribution is *deterministic*, that is the variance is null. Imposing $\sigma_S = 0$ in the previous formula we get the expression of the average response time for a $M/D/1$ queue.

3.3 Networks of Queues

Queueing Networks in general A queueing network is a system where a set of queues are interconnected in an arbitrary way. Figure 3.2 shows the

simplest queueing network, that is two $M/M/1$ queues connected in sequence. The arrival process at the latter queue is exactly the output process of the former one; thus it is more correct to identify the second queue with the notation $./M/1$ to express the fact that the arrival process at the second queue is dependent from the rest of the network.

Figure 3.2: Two $./M/1$ queues in series.

There exist different classes of queueing networks. A first distinction can be made among cyclic and acyclic networks. It is also useful to distinguish between open and closed networks. The classification is particularly useful because in literature there are several theorems that show how, for specific classes of networks, there exists the possibility of deriving a so called *product-form* solution. Solving a queueing network in product-form means that the performance of the whole system can be analytically derived in a compositional way, starting from the analysis of single queues in isolation. The key point is that a lot of different algorithms exist to evaluate the performance of product-form networks. This means that if we were able to model an architecture as a product-form queueing network, then we could apply an algorithm to extract some parameters of interest, like the system waiting time, and use them to estimate the under-load memory access latency. Unfortunately we will see that things are not so simple. In the following we explain the particularly meaningful class of closed queueing networks and we show an important result known as *BCMP theorem*.

Closed Queueing Networks In a closed queueing network there cannot be neither arrivals nor departure outside the network. Thus the population of the network is constant. Equivalently, for reasons that will be clear in the next section, we like to think at these networks as systems where *a new request is allowed to flow only when another request departs from the network*. Figure 3.3 shows the simplest closed queueing network.

For a closed queueing network it is useful to introduce the concept of *class*: all clients belonging to a specific class share the same routing politics

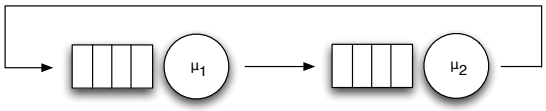


Figure 3.3: A closed system: two $./M/1$ queues in series with cycle.

at a queue. This means that clients belonging to different classes could be routed to different queues once serviced at the same queue.

We end up this overview by showing one of the main results of the BCMP theorem [5], which will be useful in the next chapter.

Theorem 3 ***BCMP networks.*** *Consider a closed queueing network in which clients can belong to different classes. Assume that all queues of the network have:*

- *FIFO service discipline;*
- *exponential service time;*

then for this kind of networks it is possible to derive a product-form solution.

This (part of) theorem is a generalization of the *Gordon-Newell* theorem [5]. The difference resides in the possibility of using classes of clients. However, notice that claiming that a product-form solution exists does not mean that it is also easy to determine it: for instance, in general, adding the client classification remarkably increase the complexity of the solution.

Chapter 4

Cost Models for Shared Memory Architectures

In Chapter 2 we have understood the importance of defining an abstract representation of a concrete architecture. This abstract architecture must be accompanied by a cost model. A cost model is fundamental to estimate the parallel application performance by taking into account both the features of the application itself, the concrete architecture and the structure of the run-time support to concurrency mechanisms. We strongly advocated that a cost model should be easy to use and conceptually simple to understand. In this perspective we have shown the idea of capturing all aspects in two simple functions T_{send} and T_{calc}. The knowledge of these function would be of invaluable importance for a programmer or (even better) a compiler to evaluate, configure and optimize parallel programs. Unfortunately, as we have already seen, shared memory architectures are heterogeneous, extremely complex systems; this fact, together with the inherent complexity of parallel programs, makes it really hard the derivation of the abstract architecture cost model, that is a good approximation for T_{send} and T_{calc}. For instance, a critical problem is to predict in which measure the limited memory bandwidth will influece the value of these parameters. To answer this question we have to estimate the so called *under-load memory access latency* R_Q, that is the average time to access the main memory when the parallel application is running. T_{send} and T_{calc} will be expressed as functions of R_Q. As far as we know, apart from [20], there are not any studies in literature addressing this topic with our methodology.

The structure of this chapter is as follows:

1. Firstly, we will formalize a general methodology to estimate R_Q in shared memory architectures. The key idea is very simple: mapping the system architecture on a Queueing Network. We will see the weaknesses of this approach that will force us to look for another approach.

2. Then we will describe a second elegant methodology based on a simpler architectural model [20]. The main feature of this approach will reside in the simple analytical resolution technique to determine approximations of R_Q.

3. Finally, we will generalize the behaviour of the latter model to accommodate a first application dependent constraint, i.e. heterogeneous processes, and we will validate the model resolution technique against experimental results.

4.1 Processors-Memory System as Closed Queueing Network

At the beginning of this chapter we pointed out the necessity of estimating the under-load memory access latency R_Q. To know this parameter is fundamental to express the cost model for an abstract architecture. In this section we show the most intuitive way to model a shared memory architecture, that is mapping it on a queueing network. Basing upon this model we will explain how, in principle, we could determine R_Q. In spite of the apparent simplicity, we will early understand that this methodology hides a lot of subtle problems.

4.1.1 Formalization of the Model

Consider a shared memory architecture in which each of the n processing node is connected, through some kind of interconnection network, to all memory modules (or equivalently to the chip's memory interface unit in case of multi-cores). A parallel application composed of n processes is being executed. The process computation alternates *think* to *wait* periods. During a

think period a process P is working on registers or data stored in its local cache. At some point a cache fault occurs and it is needed to load a block by issuing a request to the main memory. P stops working until the memory request is satisfied, i.e. until the requested block is sent back to the P's cache. The duration of this latter *wait* period, that can be strongly influenced by the workload generated by the other processes of the parallel application, corresponds to R_Q.

We can model this system with a closed queueing network, as in Figure 4.1. We identify:

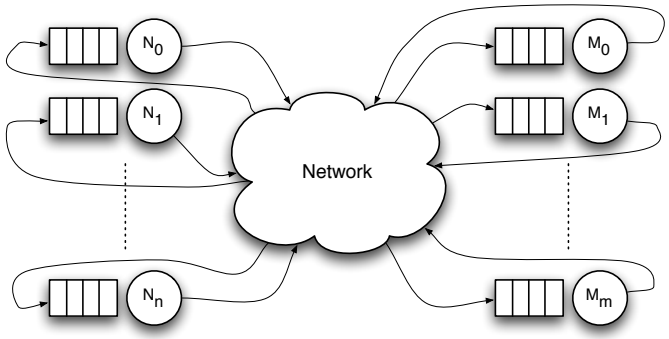


Figure 4.1: A closed queueing network model for a shared memory architecture.

- processing nodes (N), memory modules (M), interface units and other firmware units (e.g.: network routers) as queues.
- the memory requests as the unit of flow (jobs) of the network.

There are n jobs, one for each process. During a *think* period, a request resides at the processing node queue. Once the *think* period expires, a request r departs from the processing node P and is routed through the interconnection network toward the memory module queue M. Once serviced at M, r is routed back to P. Assuming that the service time at each queue is exponentially distributed, we clearly end up with a $BCMP$ network (that has a product-form solution, as pointed out by Theorem 3).

Let $Path$ be the multi-set of queues that have to be traversed by r to go from P to M and vice versa, with $M \in Path$ and $P \notin Path$. The

performance indexes we are interested in are the average times R_{Q_i} spent by r at each queue $i \in Path$. We can estimate the under-load memory access latency R_Q as:

$$R_Q = \sum_{i \in Path} R_{Q_i}$$

4.1.2 Performance Analysis of the Model

Solving the closed queueing network model of a shared memory architecture is the process of determining R_Q. First of all we need to parametrize the model, i.e. we have to fix some values of the queueing network, among which the service time at each server. We notice that:

- the service time at nodes N_i ($i = 1, \cdots, n$) corresponds to the process *think* period. Its average value T_P is a parameter principally extracted by profiling of the *sequential algorithm*. This holds for the simplest situations, anyway in chapters to come, an example in which the sequential code is necessary but it is not sufficient for deriving T_P will be presented.

- the service time at memory modules M_i ($i = 1...m$) with mean value T_S is an architecture-dependent parameter, thus it is known in advance.

There are n classes, one for each process. Each class is associated its own unique routing matrix M_i: this way it is possible to route a request to a certain memory module and, at the same time, sending back the answer to the processing node that originated it.

At this point we need an algorithm that takes as input these information and produce as output the performance metrics of the queueing network, e.g. throughput and waiting time at each server. The best algorithm to solve this kind of product-form networks is the *Mean Value Analysis* [18, 11] (MVA).

MVA allows to compute average queue lengths and response times, as well as throughputs. MVA is a conceptually simple algorithm based on two important theorems: the *Arrival Theorem* [18], that states the state of a system immediately before an arrival is independent of that arrival, and the *Little's law* 2. The time and space complexity of MVA is polynomial in systems with a single class of clients ($O(n^2)$), while it grows exponentially

with the number of classes. Since our model is a multi-class one, we could either:

1. *simplify* and *modify* our model by using a single class of clients (even changing drastically it),

2. or use different versions of the original algorithm, that go under the name of *Approximate Mean Value Analysis* techniques. These algorithms find out approximations of the expected solution mitigating the problem of exponential time complexity **??**.

At first sight we could be tempted to opt for the second solution. Exploiting a well-known algorithm to solve the model, although obtaining only an approximated solution, is an inviting perspective. In principle, we could proceed this way. However, we need to be care of the following aspects.

- **Complexity of the actual model.** Building the closed queueing network model of a shared memory architecture is not so straightforward. Figure 4.1 is just a logical scheme: it suffers from the lack of the network model, the shared memory hierarchy, the potential parallelism within the processing node and so on. Clearly, representing all these elements in our model would be nonsense because of the exceeding complexity. Therefore we need a trade-off. We advocate that at least the memory hierarchy, when shared by a set of processors, and some relevant application dependent constraints, e.g heterogeneous processes, should be modelled.

- **Importance of qualitative reasoning.** We claim that a cost model, to work, must be simple. Necessarily simple to understand, ideally simple to evaluate. The architectural model we have discussed earlier is neither of them. It is not simple to study and, moreover, it is not possible to intuitively foresee how a change in the parallel application will be reflected on the final performance, at least until a new instance of the MVA algorithm will be executed. We would like an analytical model, e.g. some kind of simple equations, that help us in understanding, for instance, how R_Q varies as a function of T_P or T_S. Unfortunately, it is extremely complex to derive such equations from the actual model, even with a deeper knowledge of Queueing Theory.

- **Flexibility of the model.** The exponential distribution is often a good approximation for our purposes, but not always. There can be cases in which a server is in reality a deterministic one, e.g. a memory module that takes a constant time to retrieve the desired information. The problem is that if we release the assumption of exponential service time, the $BCMP$ theorem (3) does not hold any more. In general, networks with servers having service times different than the exponential one cannot be reduced in product-form. In this case the stochastic modelling of the system is no more a Markov process. This is perhaps one of the biggest problem in performance modelling, and not only in our context. There exist approximated versions of MVA, based on heuristics, that try to solve these limitations, but results are often not as good as expected. Nevertheless, our experience suggests that it is very common to encounter new scenarios in which MVA either is not sufficient to solve our model (because its assumptions are violated) or requires a partial redesign to accommodate our necessities.

In light of these considerations, instead of increasing the complexity of building a shared memory architecture cost model by studying how to use MVA techniques for our complex architectural models, we prefer to *simplify* the original model and looking for new, easier ways of computing its performance measures.

In the next section we will show a simplified version of this model. However, some of the concepts that we have introduced will be exploited as well in the following.

4.2 Processors-Memory System as Client-Server Model with Request-Reply Behaviour

4.2.1 Formalization of the Model

Consider a system in which a set of N client modules $C_1, C_2, ..., C_N$ send requests to a server module S and need to wait for an explicit reply in order to continue their elaboration. An example of this scheme is shown in Figure 4.2. Notably cases of this interaction pattern are some client-server

parallel applications as well as processors-memory systems. Therefore, the model formalized in this chapter may be applied to a lot of different domains and abstraction levels. Our main goal in a client-server model with request-reply behaviour is to estimate the average response time R_Q of S.

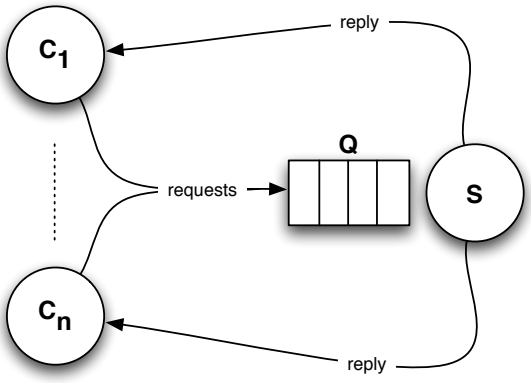


Figure 4.2: Client-server system with request-reply behaviour.

A *logical* queue Q is present in front of S. We talk about a logical queue because conflicts for resource contention could happen not only in S, but also nearby other modules that, for complexity reasons, are abstracted away from the system. For example, think to a scenario in which client's messages need to travel along an interconnection network to reach S; it is unlikely that the network is a crossbar, thus the probability $p_{conflict}$ that a request has to be queued somewhere in the network is different than 0. Depending on the value of $p_{conflict}$ the cost model can be properly parametrized to take care of such conflicts. For example, a very simple yet meaningful approach consists in increasing the latency of the server S. In these cases we may say that S is logically the subsystem that includes both the interconnection network and the module that carries out clients' requests.

We instantiate the model on a generic multiprocessor system with N processing nodes and m shared memory macro-module by using the same methodology of [20]:

- Let p be *the average number of processing nodes sharing the same memory macro-module*. It is very important that p is as low as possible in order to minimize the congestion overhead at a memory macro-module.

In an SMP architecture, in which memory accesses are uniformly distributed over m macro-modules, p can be estimated as the mean of the binomial distribution $p = \frac{N}{m}$. In a NUMA architecture, the uniform distribution does not hold any more, but the value of p is dependent on specific characteristics of the parallel application. It has been shown in [20] that, for structured parallel programming, there exist optimum strategies and heuristics to map processes onto processing nodes in such a way to minimize the value of p.

- The clients $C_1, C_2, \cdots, C_p$ model p processing nodes everyone executing a process and sharing the same macro-module of main memory
- Initially, we assume for simplicity that all the clients have an identical behaviour, but in the future we will remove this hypothesis. Further, we will assume that the behaviour of a client is the one described in the previous chapter and nothing more complicated. So *think* periods alternate to *wait* ones and the duration of a *think* period is represented by an exponentially distributed random variable, with mean value T_P.
- the server S models the shared memory macro-module (*and potentially* even the interconnection network among the processing nodes and the memory macro-module). Its service time is assumed to be exponentially distributed with mean value T_S

4.2.2 Assumptions

The cost model for determining R_Q in shared memory architectures implies a complex evaluation due to a large number of degrees of freedom. We have already seen that a lot of problems arise when trying to model the architecture as a general closed queueing network. With the client-server approach we remarkably mitigate the complexity:

1. *by simplifying the original model.* The complex closed queueing network shrinks to a single queue model. Processing nodes become simple modules that generate requests with a certain frequency. The focus is on a single memory macro-module rather than the whole memory system. The network model is cut away. However, network conflicts overhead may be taken into account during the resolution of the model.

2. *by using a simple yet meaningful analytical resolution technique* that we will study in the next section.

Since we are seeking for a resolution technique characterized by reasonable complexity and, at the same time, that is able to retrieve approximated results, we need to rely on some further assumptions and simplifications:

- we have already said that T_P is the mean value of an exponentially distributed random variable. Actually this distribution depends on the parallel application characteristics, and could be even different from the exponential one. For instance, when the elements of an array are read linearly and the computation between one read and the subsequent takes always the same amount of clock cycles (which is quite common in a program), then the proper distribution should be the deterministic one. However, we are rather interested in evaluating the inter-arrival time at S. Since we are assuming independent processing nodes, the input stochastic process at S shows a random behaviour that can be *approximated* by an exponential distribution. In the following we will assume this distribution for the input stochastic process at the server but, as we will mention, there exist improvements from this point of view.

- we focus on the server service time T_S. The behaviour of a memory macro-module is in most of the cases deterministic [20]. That is, any request generated by a processing node takes always the same amount of clock cycles to be served. On the other hand there are also memory systems that exhibit a non-trivial behaviour. Some kind of memories are able to exploit space and time locality of groups of consecutive requests for elements stored on the same row of a memory bank [1]. In these cases, T_S is not more a constant so an exponential distribution could be used as a better approximation. Other memories, e.g. DDR-2 memories, have a *load-dependent* behaviour: the more the number of request in Q, the lower is the average service time. This is because requests can be reordered in such a way to exploit the aforementioned locality properties. For instance, the Tilera Tile64 is characterized by a load-dependent memory and studies about this behaviour are reported

in [15]. For our purposes, we will focus only on server service time exponentially distributed.

- we are assuming homogeneous clients. However, processes of a parallel application can exhibit different behaviour each other, but in general this does not hold in case structured parallel programming approach is used. At first sight, from the cost model point of view two processes should differ if their *think* period is different, i.e. they have different service time T_P. However, a formalization taking into account the structure of the parallel application will be given in Chapter 6 in order to recognize heterogeneous processes in an easier way.

- another important assumption that we are making is that clients are characterized by only one T_P for all their life cycle. Taking into account structured parallel applications, this way to model does not reflect properly the behaviour of processes. A very common example could be a process starting with a computational phase (the so called *think* period) that will be followed by an inter-process communication (for instance a *send*). When this last primitive will be executed, the service time of the process during that phase can vary a lot from the computational one, e.g. even an order of magnitude. Informally, we can define a *process phase* a lapse of time of the process characterized by a certain average service time. Having clients characterized by only one T_P, a first way to take into account phases could be to use directly the overall weighted average service time. We will study the impact of this phenomenon in Chapter 6, in the meanwhile we will assume only one phase.

- we are modelling non-hierarchical systems, i.e. architectures where only the main memory is shared. Instead, especially in state of the art and upcoming multi-cores, the trend is to provide cores with shared levels of caches (see Section 2.1). The intuition is that concurrent accesses to shared resources can introduce a significant overhead, especially if the number of sharers is large. The problem of hierarchical architectures is addressed in Chapter 7.

4.2.3 Model Resolution

In [20] it is shown that the following system of equations captures the behaviour of the client-server model with request-reply behaviour.

$$\begin{cases} T_{cl} = T_P + R_Q \\ R_Q = W_Q(T_s, T_A) + t_{a0} \\ \rho = \frac{T_S}{T_A} \\ T_A = \frac{T_{cl}}{p} \\ \rho < 1 \end{cases} \tag{4.1}$$

Each client generates the next request only when the result of the previous one has been received. The behaviour of a client, as we have already said, is cyclic: *think* periods (T_P) alternates to *wait* ones (R_Q), leading to a certain client average inter-departure time T_{cl}. This fact is captured by the equation $T_{cl} = T_P + R_Q$. Once we know T_{cl} we can determine the server average inter-arrival time T_A as $\frac{T_{cl}}{p}$ applying the Theorem 1. Finally, the under-load memory access latency R_Q is simply given by the average waiting time W_Q plus a constant known in advance that is the base latency t_{a0}. As already told, if it is necessary, t_{a0} also contains the impact of the network. The expression of W_Q depends on the type of Q as we mentioned in Section 3.2.

The system has a self-stabilizing behaviour: e.g. a temporary increase of T_A has the effect of decreasing R_Q, that in turn tends to lower T_A itself since T_{cl} will decrease. This is also an example of qualitative reasoning. Since the system shows a self-stabilizing behaviour, it could be proved through Markov analysis that $\rho < 1$. This means that a steady-state solution exists.

Assuming that Q is either $M/M/1$ or $M/D/1$, solving this system with respect to R_Q leads to a second degree equation in ρ. The two solutions ρ_1 and ρ_2 are always such that $\rho_1 < 1$ and $\rho_2 > 1$, thus the solution of the model must be subjected to the constrain $\rho < 1$.

Although suffering from limitations of the previous section, this model resolution technique is very interesting because:

- it is simple

- it is based on mean values quantities rather than probability density functions, and this further simplifies the analysis. We advocate that a resolution technique based on mean values is sufficiently accurate for our purposes.

- it enables qualitative analysis

- it is good also for quantitative analysis, i.e. it gives quite good approximation to the real value of R_Q as reported in [20]. Moreover, removing some assumptions, different resolution techniques are possible improving the quality of the analysis [15].

- it is parametric in the service times distribution. The formula of the average waiting time W_Q is chosen according to the scenario we are modelling. For example, if the memory subsystem shows a deterministic behaviour, than we will use the standard Queueing Theory formula for $M/D/1$ queues (Section 3.2).

- it is prone to generalization and accommodations. For example, we can easily deal heterogeneous clients with a more general variant of the model or we can take into account the impact of the interconnection structure directly in the latency t_{a0}

Besides the distribution of the server service time, another important aspect is the choice of the parameter of this distribution, that represents the frequency $\mu = \frac{1}{T_S}$ at which requests are carried out. Consider the two extreme cases, which are also the most important ones:

- $T_S = T_{a0}$. The server service time is the base latency. In this case we model the system as if the network between clients and server would be a bus. It is known that a bus can handle only one request at a time; this behaviour would be captured by our system, since client requests would be blocked immediately in the processing node.

- $T_S = T_M$, being T_M the average time required by a memory macromodule to carry out a request. This is the case wherein network conflicts are neglected. This assumption is meaningful in particular conditions as explained in Section 2.3.1. In particular, we recall that this

holds for fat trees and networks on chip (unless the network is a bus) because, in the formers, the time needed by a request to be routed within the network on chip is significantly lower than the one spent nearby the memory (queueing delay plus service time). In the following, we will assume $T_S = T_M$.

Finally, remember the original goal: we are determining the cost model of an abstract architecture. The abstract architecture is characterized by two functions: T_{send} and T_{calc}. To express these functions, we had to understand the system ability to execute a certain amount of instructions *in presence of memory conflicts*, that is the real bandwidth of processing nodes. In this perspective, the value of R_Q will be used to express such functions, as shown in [20].

4.3 A variant of the Client-Server Model: Heterogeneous Clients

In this section we propose a variant of the client-server model in which heterogeneous clients are considered. The reason for doing that is to model the behaviour of a shared memory architecture taking into account a first impact due to the parallel application executing on it.

As explained above, processes alternate *think* to *wait* periods. Until now, we were assuming for simplicity that *think* periods were always the same for all processes but this does not hold in some cases. Some examples are classical farm or map paradigms because they are composed by the so called service processes, e.g. emitter and collector, in addition to workers. However, it is worthwhile to say that the impact of emitter and collector is negligible if the number of workers is high enough. We recall that in general homogeneous processes are involved because the structured parallel programming approach.

The fundamental characteristic of a process is its inter-departure time T_{cl}, i.e. the average time between two consecutive memory requests toward a given macro-module. This value is principally affected by T_P and R_Q. Being R_Q (once found) the same for all processes, the only way to distinguish a process from another one is its service time T_P.

4.3.1 Definition

In order to deal this topic, heterogeneous clients should be used in the model. If we look at 4.1, we can recognize that the equations system is prone to accomplish heterogeneity. In fact, the original version is only a specific case in which clients are homogeneous. In the more general case, we can explicitely write the inter-departure time T_{cl} of the p clients sharing the server obtaining the following system of equations.

$$\begin{cases} T_{cl_1} = T_{P_1} + R_Q \\ \dots \\ T_{cl_p} = T_{P_p} + R_Q \\ R_Q = W_Q(T_s, T_A) + t_{a0} \\ \rho = \frac{T_S}{T_A} \\ T_A = \frac{1}{\sum_{i=1}^{p} \frac{1}{T_{cl_i}}} \\ \rho < 1 \end{cases} \tag{4.2}$$

The equations system 4.2 describes the model generalization in a natural way, nevertheless the resolution complexity, i.e. the degree equation in ρ, increases if the number of different T_P in the system increases.

4.3.2 Comparison against the Queuing Network Simulator

In this paragraph we want to compare the resolution technique 4.2 against the outcome provided by the queuing network simulator *Java Modelling Tools* (JMT) [4]. The modelled scenario has the following features:

- the number of clients is fixed to $p = 16$; seven of them have a certain service time T_{P_1}, other seven a T_{P_2} while the last two have a fixed $T_P = 100\tau$. The idea is to simulate a functional partitioning with independent workers and two service processes, i.e. a dispatcher and a gather. Further, we are assuming that the number of workers is the

optimal one in such a way we can assume them (practically) always working.

- the distribution is exponential for all service times. Since p is fixed, the service time of clients is the degree of freedom. In particular, in each test T_{P_1} will be *fixed* to a certain value chosen in the range $[100\tau - 800\tau]$ while T_{P_2} will *vary* in the same range in such a way will be possible to find results for different load states of the server. An important consideration is that since p is fixed to 16, T_P values higher than 800τ are not such much interesting because the server will be in average unloaded.

It is worthwhile to say that the most of the values utilized for instantiate the parameters have been influenced by our studies on the Tilera Tile64 multi-cores [2]. So we have:

- the average server service time is $T_S = 29\tau$ because this value is typical of macro-modules of DRAM2 memories. As discovered, the memory in the Tilera Tile64 is load-dependent but, for our purposes, an exponential distribution will be a good assumption. A more in depth study about the load-dependent behaviour is present in [15].

- the client service time range is typical of processes in computation phase on shared memory architectures, like the Tilera Tile64. More difficult is to establish the right value for service processes. In fact, while we can practically assume workers always working, this does not hold in general for emitter and collector. Therefore, it may happen that these processes alternate *stopping* periods to communication and how much time they stop before a communication influences the value of T_P that, in this case, can not only be extracted by profiling but depends also by other factors. We will introduce this problematic in Chapter 6. In the meanwhile, we use the constant 100τ as first approximation for the service time of these clients just for evaluating the theoretical impact of emitter and collector in case the number of workers is not such much high.

- the base memory access latency is $t_{a0} = 72\tau$ evaluated on the Tilera Tile64 taking into account the latency of the memory (29τ) plus the

base network latencies for the memory request and memory reply. These last two values depend by the distance travelled in the mesh and by the length of the firmware messages according to the methodology explained in 2.3.2

We will list the results in the following order:

- the graphs in Figure 4.3, 4.4 and 4.5 show the progress of RQ with T_{P_2} varying on x axis. In each graph T_{P_1} is fixed and its value is specified in the name of the shape visible in the legend. Each name is composed by two parts: the former specifies the source, i.e. the simulator (SIMULATION) or the analytical resolution (CS), while the latter contains the value of T_{P_1}, that could be 100τ, 300τ or 500τ. As mentioned above, higher T_P values do not introduce substantial error so they are not reported.

- the graphs in Figure 4.6 and 4.7 show the absolute and relative error of the analytical resolution against the simulation. We do immediately an important preliminary observation: the absolute error as such does not take into account that at different T_{P_1} correspond different RQ shapes. Instead, the relative error correlates the absolute error with the right R_Q shape. Having said this, we will base on relative error for following comments.

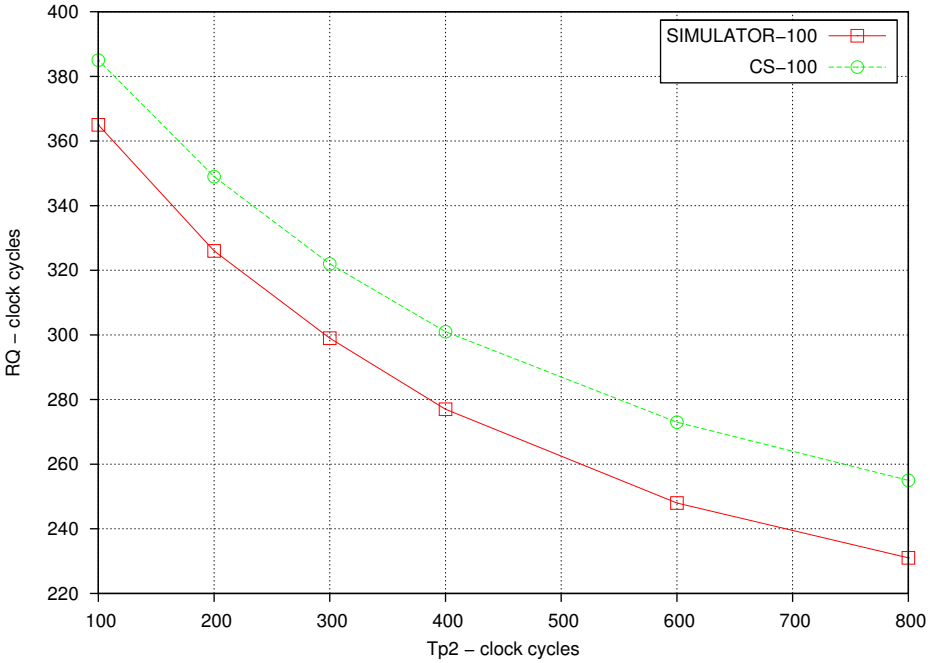


Figure 4.3: The R_Q shapes for T_{P_1} fixed to 100τ

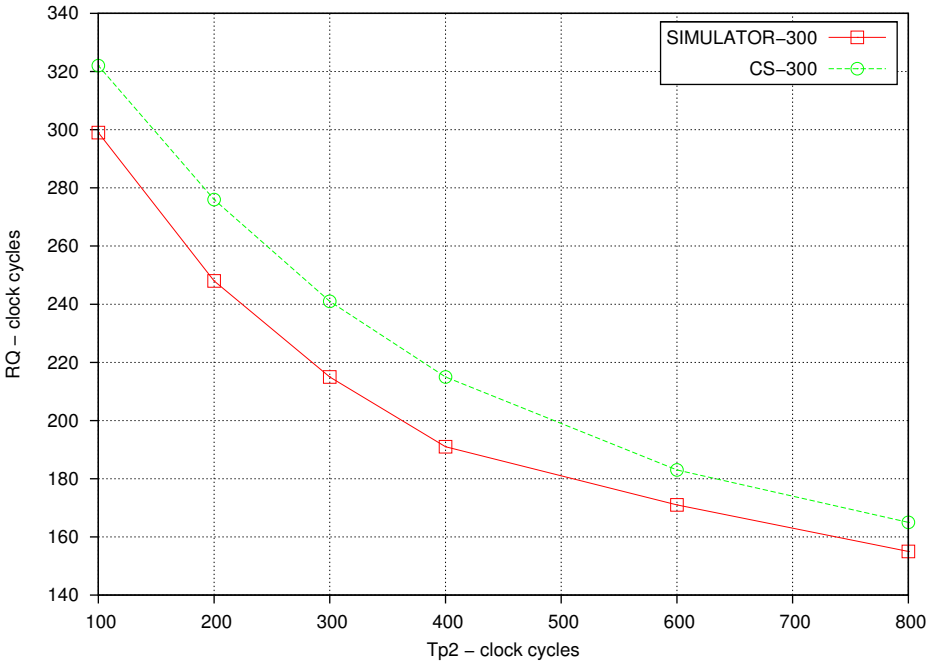


Figure 4.4: The R_Q shapes for T_{P_1} fixed to 300τ

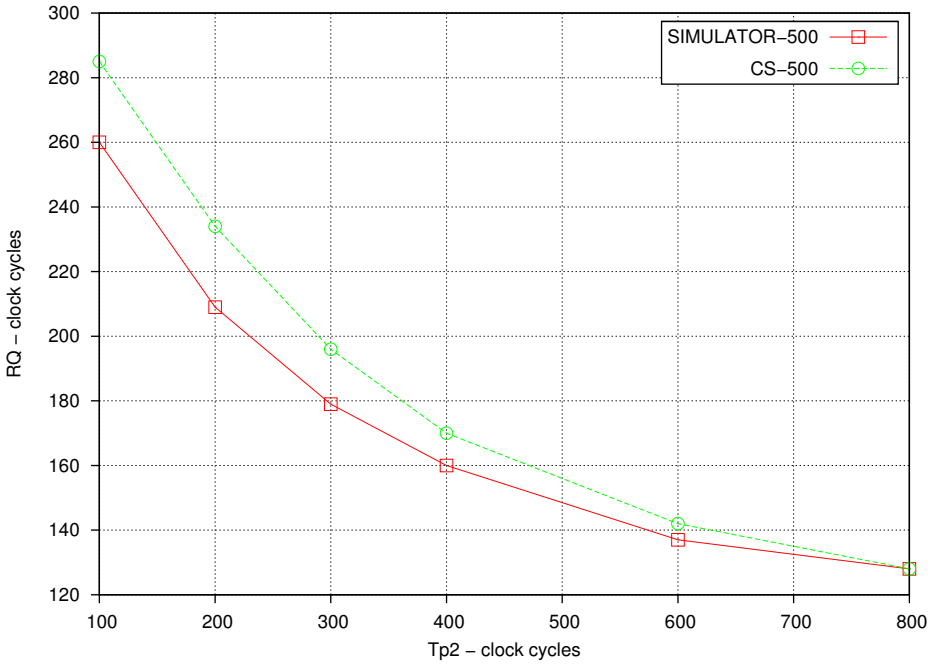


Figure 4.5: The R_Q shapes for T_{P_1} fixed to 500τ

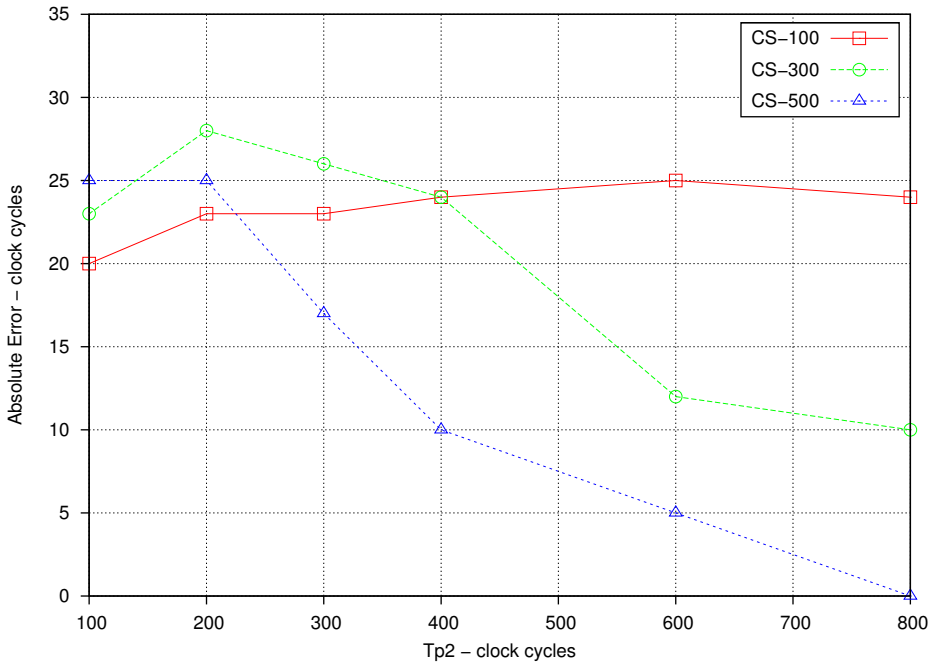


Figure 4.6: Absolute Error

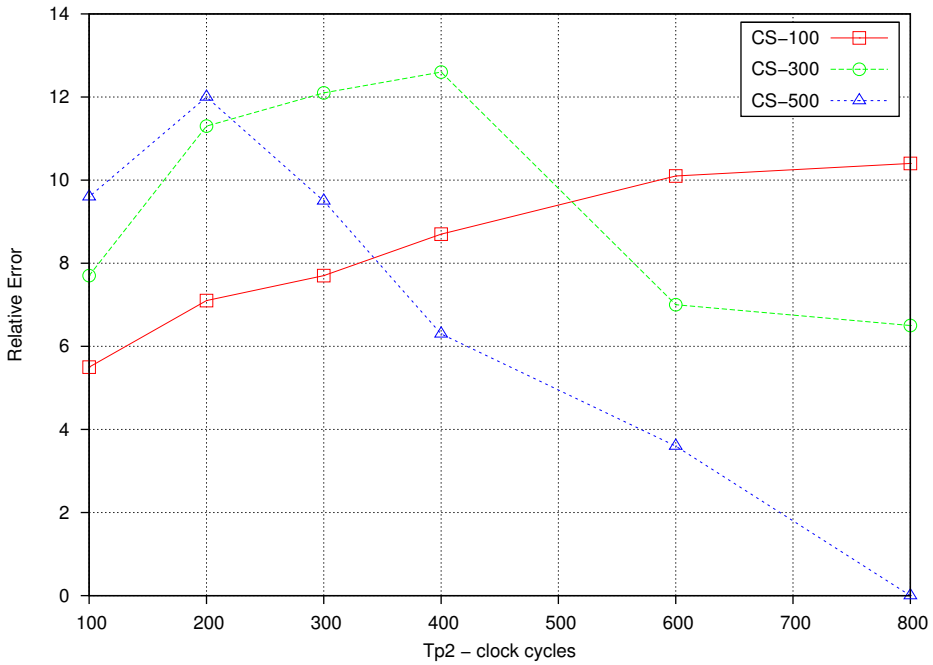


Figure 4.7: Relative Error

Figure 4.8: Errors of analytical resolution technique against the simulation for the three cases $T_{P_1} = 100\tau$, 300τ and 500τ

4.3.3 Comments

In light to the above results, we have the following comments:

- first of all, we notice that there is a gap between the RQ shape of the simulation and the one resulting from the resolution technique. Further, the approximation of the resolution technique is slightly better when the server is not much loaded as well visible in Figure 4.4 and 4.5. In other words, the difference tends to smooth for high T_P values. This behaviour is close to the one with all homogeneous clients [20] so the reason of this is not strictly related to having heterogeneous clients, but it something intrinsically to model assumptions. Since the population in the system is finite and fixed, the inter-arrival process at the server can not be exponential. Moreover, clients receive a feedback from the server that influences their service time and , consequently, their inter-departure time. From this point of view, some model improvements have been done and different resolution techniques present in [15] give better approximations under certain conditions. The possibility and the goodness of using these improvements with heterogeneous clients have to be verified yet.

- the just mentioned model approximation is the major reason for the gap between the simulator shape and the resolution technique shape but it is not the only one. Looking at the Figure 4.7, we focus on the CS-100 and CS-300 shapes for $T_{P_2} = 100\tau$. We have that the relative error in case *all* the clients have a service time equals to 100τ is 5.5% (CS-100) while it is 7.7% if seven of them have a service time equals to 300τ. Hence in this case the relative error is greater in a situation in which the server receives less requests. This can be explained as the impact to having heterogeneous clients. Notice that the same thing also happens for CS-500 and the relative error is bigger with respect to the other two (9.6%). So we can conclude that when the difference between the service time of different clients grows, the error of the resolution technique also increases.

4.4 Conclusions

The estimation of the under-load memory access latency is the first step in order to derive the abstract architecture cost model (T_{send} and T_{calc}), by means of which the physical system can be completely abstracted. We will not go into the details of the formal derivation of these functions. The interested reader is invited to consult [20] for more details. However, it is sufficient to know that both T_{send} and T_{calc} are functions of R_Q, therefore a meaningful estimation of its value is crucial.

We have shown that a solution based on pure Queueing Network Theory is infeasible from the complexity point of view. We prefer the client-server model and its simple analytical resolution technique based on a minimal set of Queueing Theory concepts. With this approach we also enable qualitative analysis which is extremely important in our context: for example, we can understand the asymptotic behaviour of a certain measure (e.g. T_{cl}, R_Q, ...) as a function of other model parameters (e.g. T_P) by taking into account even the feedback effect. The importance of qualitative reasoning has been shown in [20].

From the quantitative analysis point of view, we have seen that a gap between the simulation and the result of resolution technique exists, and it is mainly due to intrinsic original assumptions. This topic has already been studied in our research group and formalized in [15] where is shown that improvements under certain conditions are possible and new resolution techniques are more accurate with respect to the original one.

In spite of this, the accuracy of this resolution technique worsen with respect to the simulation if we start to consider parallel application constraints like heterogeneous processes. A possible use of at least one of the just cited more sophisticated techniques could be taken into account, but until now the goodness of the accuracy is not predictable. Further, it is worthwhile to say that when the number of different clients increases, the equations degree in the system 4.2 increases raising the analytical resolution complexity. On the other hand, we can assume a limited number of different clients in the majority of the applications resulting from a structured parallel programming approach.

Although the quality of the results is quite good, in order to apply this

model to real architectures we need to take care *at least* of the following aspects:

1. the gap between the analytical resolution and the simulation should be in general decreased in order to have a (more) precise cost model.

2. the client-server model cannot be applied to hierarchical systems as it stands

3. the analytical resolution of the client-server model is prone to accomplish *some* constraints given by the parallel application but the structure and the specific characteristics of the parallel application are not taken completely into account yet

4. modelling queues different than the basic ones ($M/M/1$, $M/D/1$) is quite difficult with this resolution technique

In light of these elements, it is necessary to further extend the model. The price to pay for extending the model is given by the necessity of improving the actual analytical resolution technique. Unfortunately, the complexity of the problem notably increases. Therefore, in the next chapter we will study the potential advantages coming from the employment of numerical resolution techniques. It is left as an open problem to find out *approximate* yet *meaningful* analytical resolution techniques modelling the aforementioned topics.

Chapter 5

Stochastic Process Algebra Formalization of Client-Server Model

In the previous chapter we have determined an elegant cost model for the under-load memory access latency in shared memory architectures. Unfortunately, a lot of problems and assumptions may impair the analytical resolution technique of the client-server model. We pointed out three key aspects.

- **Probability distributions and queue types.** Inter-arrival and service times at a queue of a client-server system may not be exponentially distributed. It is well-known that providing analytical resolution techniques for non-Markov system is a non-trivial task. Anyway, we know from [20] that a good approximation for deterministic service times (using the $M/D/1$ queue) can be achieved with 4.1, but we can not say the same about the modelling of:

  1. deterministic inter-arrival times
  2. distributions different from the deterministic and exponential ones (if they were needed)
  3. queues exhibiting a load-dependent behaviour

- **Impact of the parallel application.** In the classic model, clients are assumed homogeneous with a fixed ideal inter-departure time T_{cl}. However, we have seen that clients can behave differently each other

for modelling heterogeneous processes. Moreover, we have already told about the alternation of different processes phases. A process itself can have a complex behaviour and modelling it through the mean value of a probability distribution could be a right approximation or not. In the simplest scenario a phase of sequential elaboration is followed by a phase of communication (either point-to-point or collective); it should be clear that the load generated on the memory system in these two phases may be drastically different, e.g. even an order of magnitude.

- **Hierarchical shared memory.** If the multi-cores trend follows the direction that has been taken, hierarchical shared memory will be a relevant feature. In these architectures, more than one level of memory hierarchy is shared by processing nodes (see Section 2.1). Therefore, conflicts for accessing shared resources could become significant for what concerns the under-load memory access latency. Somehow we need to measure also these kind of conflicts; perhaps enhancing the client-server structure to model a hierarchy of servers (*hierarchical client-server systems with request-reply behaviour*).

It is obvious that if we want to take care of these aspects, the resolution techniques should be adequately improved. Again, the problem is to determine the trade-off between the complexity of the resolution technique and the quality of the approximated results. In light of this, the following methodology is proposed:

- the client-server model with request-reply behaviour remains the reference paradigm (where needed, server may be structured on a hierarchy),
- but *numerical* resolution techniques will be used to evaluate the under-load memory access latency, in place of the analytical ones.

We advocate that the employment of numerical techniques can overcome the complexity deriving from the formalization of analytical ones. The idea is to describe the client-server model at the level of Markov Chains. There are a lot of direct solution methods for moderately sized Continuous Time Markov Chain (CTMC) models, while iterative techniques exist for huge sized models [9]. Since Markov processes can be difficult to construct by hand

(this holds for human beings, but maybe not for a compiler), we will exploit as intermediate description language a *stochastic process algebra* (SPA). An SPA approach is very intriguing because the aforementioned aspects may be addressed with a formal and structured approach.

The structure of the chapter is the following:

1. firstly, we introduce and describe the stochastic process algebra PEPA
2. secondly, we show how to express a basic client-server model with the new formalism
3. finally, the accuracy of the new resolution technique will be compared against experimental results

Hierarchical systems and a methodology for an in-depth analysis of the parallel application impact will be formalized in the next chapter following this approach.

5.1 PEPA: a Process Algebra for Quantitative Analysis

Performance Evaluation Process Algebra [10] (PEPA) is a high-level description language for Markov processes which belongs to the class of *Stochastic Process Algebras* [6] (SPA). Among the wide class of SPAs, we choose PEPA because it is *simple* but at the same time it has sufficient expressiveness for our purposes. The simplicity comes from the structure of the language: PEPA has only a few elements and a formal interpretation of all expressions can be provided by a structured operational semantics. In this section we just introduce the minimal set of PEPA features strictly necessary to model client-server with feedback systems; for a deeper understanding the reader is invited to consult [10].

We recall that Markov processes rely on the memoryless property of the exponential distribution.

Definition 4 ***Markov Process.*** *A stochastic process $X(t)$, $t \in [0, \top)$, with discrete state space S is a Markov process if and only if, for $t_0 < t_1 < ... <$*

$t_n < t_{n+1}$, the joint distribution of $(X(t_0), X(t_1), ..., X(t_n), X(t_{n+1})$ is such that

$$Pr(X(t_{n+1} = s_{i_{n+1}} | X(t_n) = s_{i_n}, ..., X(t_0) = s_{i_0}) =$$

$$Pr(X(t_{n+1} = s_{i_{n+1}} | X(t_n) = s_{i_n})$$

Intuitively, this means that the probability of X to go into the state $s_{i_{n+1}}$ at time t_{n+1} is *independent* of the behaviour of X *prior* to the instant t_n or, in other words, it depends *exclusively* by the state s_{i_n} of X at time t_n. It is important to keep in mind this property when working with PEPA.

The Language A PEPA system is described as the composition of *components* that undertake *actions*. Components correspond to identifiable parts in the system. For instance, in our context, clients and servers will be the components of the systems. A component may be atomic or may itself be composed by components. The language is indeed *compositional* in sense that new components may be formed through the cooperation of other ones. Each component can perform a finite set of actions. An action has a duration (or delay) which is a random variable with an *exponential distribution*. Consequently, the *rate* of the action is given by the parameter of the exponential distribution. For example, the expression

$$P \stackrel{def}{=} (\alpha, r).Q$$

represents the definition of a new component P which can undertake an action α at rate r to evolve into another component Q (defined somewhere else). Since the duration of all actions of the system are exponentially distributed, it is intuitive to say that the stochastic behaviour of the model is governed by an underlying CTMC.

The syntax of the PEPA language is formally defined by the following grammar.

$$\begin{array}{rcl} S & ::= & (\alpha, r).S \mid S + S \mid C_S \\ P & ::= & P \underset{\mathcal{L}}{\bowtie} P \mid P/L \mid C \end{array}$$

S denotes a *sequential component* and P denotes a *model component* which executes in parallel. C and C_S stand for constants to denote either a sequential or a model component (the effect of the syntactic separations is to allow

to build only components which are cooperation of only sequential components, which has been proved in [10] to be a necessary condition for building *ergodic* Markov processes, i.e. amenable to steady-state analysis).

Prefix

$$\frac{}{(\alpha, r).E \xrightarrow{(\alpha,r)} E}$$

Choice

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E + F \xrightarrow{(\alpha,r)} E'} \qquad \frac{F \xrightarrow{(\alpha,r)} F'}{E + F \xrightarrow{(\alpha,r)} F'}$$

Cooperation

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,r)} E' \underset{L}{\bowtie} F} (\alpha \notin L) \qquad \frac{F \xrightarrow{(\alpha,r)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,r)} E \underset{L}{\bowtie} F'} (\alpha \notin L)$$

$$\frac{E \xrightarrow{(\alpha,r_1)} E' \quad F \xrightarrow{(\alpha,r_2)} F'}{E \underset{L}{\bowtie} F \xrightarrow{(\alpha,R)} E' \underset{L}{\bowtie} F'} (\alpha \in L) \quad \text{where } R = \frac{r_1}{r_\alpha(E)} \frac{r_2}{r_\alpha(F)} \min(r_\alpha(E), r_\alpha(F))$$

Hiding

$$\frac{E \xrightarrow{(\alpha,r)} E'}{E/L \xrightarrow{(\alpha,r)} E'/L} (\alpha \notin L) \qquad \frac{E \xrightarrow{(\alpha,r)} E'}{E/L \xrightarrow{(\tau,r)} E'/L} (\alpha \in L)$$

Constant

$$\frac{E \xrightarrow{(\alpha,r)} E'}{A \xrightarrow{(\alpha,r)} E'} (A \stackrel{def}{=} E)$$

Figure 5.1: Structured Operational Semantic of PEPA.

The structured operational semantic is shown in Figure 5.1. Below an intuitive description of most used PEPA operator is provided. For a complete treatment the reader is invited to consult [10].

- **Prefix ($(\alpha, r).P$)** This is the basic mechanism to express a sequential behaviour in PEPA. As already said, a component performs an action α at rate r behaving subsequently as P.

- **Choice ($P+Q$)** This operator represents a component that may behave either as P or as Q. Assume that α and β are the actions that enable respectively P and Q, characterized by their own rate. The idea behind the Choice operator is that once an action has been completed, the other is discarded. For instance, if the first action to be completed is β then the component moves to Q, "forgetting" the other branch.

- **Cooperation ($P \underset{\mathcal{L}}{\bowtie} Q$)** This operator denotes the cooperation between P and Q over L. L is the cooperation set that contains those activities on which the components are *forced* to synchronized. The rate of this shared activity has to be altered to reflect the slower component in the cooperation (see how in Figure 5.1). It is important to notice that for actions not in L components proceed *independently* and *concurrently* with their enabled activites. Actually cooperation is a *multi-way synchronization* since more than two components are allowed to jointly perform actions of the same type. When concurrent components do not have to synchronize the cooperation set L is empty; in these cases we will use the abbreviation $P||Q$ to denote P and Q running in parallel. We will use also a simple syntactic shorthand to denote an expression like $(P||P||...||P)$ as $P[N]$, with N the number of times that P is replicated. Finally, we point out that there can be situations in which two components do synchronize, but the rate of the shared activity is determined by only one of the component in the cooperation. In this case the other component is defined as *passive*. The rate of the activity for the passive component will be denoted with the symbol $\top$.

5.2 A PEPA Formalism for Client-Server Model with Request-Reply Behaviour

5.2.1 Definition

A PEPA program for the classical client-server model with request-reply behaviour (Section 4.2) can be instantiated to model a processors-memory system just knowing the following parameters:

- T_P, the mean time between two consecutive accesses of a processing node (executing a process) to a certain memory macro-module
- T_S, the average service time of that memory macro-module
- p, the average number of processing nodes accessing that memory macro-module
- T_{req}, the base network latency for a memory request
- T_{resp}, the base network latency for a memory reply

The resulting PEPA program is shown below.

$$
\begin{aligned}
r_{request_c} &= 1.0/T_P \\
r_{reply} &= 1.0/T_S
\end{aligned}
$$

$$
\begin{aligned}
Client_{think} &\stackrel{def}{=} (request, r_{request}).Client_{wait} \\
Client_{wait} &\stackrel{def}{=} (reply, \top).Client_{think}
\end{aligned}
$$

$$
Server \stackrel{def}{=} (request, \top).Server + (reply, r_{reply}).Server
$$

$$
Client_{think}[p] \underset{request,reply}{\bowtie} Server
$$

Each client models a process (on a processing node) that operates forever in a simple loop, completing in sequence the two phases *think* and *wait* (Figure 5.2).

As already told, the length of the *think* phase is T_P. At the end, a *request* action is executed and the client waits for a *reply*, i.e. it starts the *wait* phase. The *request* action is a shared action between the clients

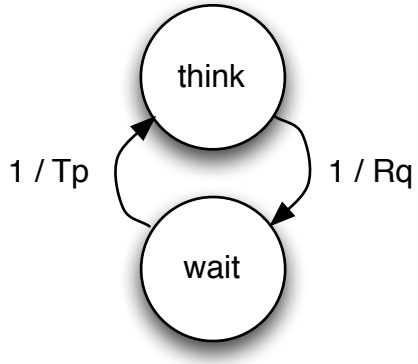


Figure 5.2: A Client alternating *think* phase to *wait* ones.

and the server and it models the situation in which a client sends a request and the server receives it. The length of the *wait* phase is R_Q. For this reason, the time needed to complete the *reply* action (phase *wait*) is initially *unspecified*. In fact, it will be imposed in another PEPA expression through the cooperation with another component. Therefore, *Client* components see *reply* as a pure synchronization operation.

The server modelling the memory macro-module can either accept a request from one of the p clients (action *request*) or send them a *reply*. The time to complete a *request* action is obviously unspecified because it depends on clients. The action *reply* is shared to model the fact that a client can go back to the *think* phase as soon as the server has handled its request.

Finally, the last expression instantiate a client-server model with p clients running in parallel that try to synchronize themselves with the server through the cooperation set containing both the two shared actions *request* and *reply*.

It is useful to highlight that even simpler solutions could be formalized: for instance, the synchronization on the action *request* is not strictly necessary. However we decided to keep it for two reasons. First, it helps to understand the semantic of the whole system (the "request-reply behaviour"). Second, it will be necessary anyway in further extensions of this basic model.

5.2.2 Quantitative Comparison with respect to other Resolution Techniques

Preliminary Considerations Solving a PEPA model means solving the underlying ergodic CTMC, i.e. computing the steady-state. We wrote and solved PEPA models using the classic tool PEPA Workbench [19]. This tool provides a lot of different numerical resolution techniques to solve the model. Different techniques can be employed depending on the size of the resulting CTMC: if the number of states is huge (hundreds of thousands) iterative yet approximate techniques are preferred. However, the models that we treat are extremely small (they never exceed a hundred of states) thus the steady-state has been directly computed employing a very standard algorithm. In all other cases, e.g. when the number of clients significantly grow, a phenomenon known as *state space explosion* may arise. However, thanks to the natural structure of our models, we may take fully advantage from both *state-reduction* and *fluid-approximation* techniques [8]. Briefly, these techniques aim to solve the state space explosion by exploiting potential symmetries in the CTMC. The presence of symmetries can be informally deduced looking at the PEPA expressions: for instance, in our model the set of homogeneous clients ("$Client[p]$") induces replicated sub-Markov chains in the underlying CTMC. These replicated subsystems will be exploited to restructure the CTMC itself and lowering the state space size.

Model Resolution Once found, steady-state information are exploited to derive the average response time $R_{Q_{server}}$ of the server. In particular these information include:

- the average population size of a state
- the throughput of the actions

In our client-server model we are interested in the average number of clients that reside in state $Client_{wait}$ (p_{wait}) and in the throughput of the action $reply$ (λ_{reply}). Indeed, by applying the Little's law (2), we can extract the average time that a client stays in the state $Client_{wait}$, which actually corresponds to $R_{Q_{server}}$:

$$R_{Q_{server}} = \frac{p_{wait}}{\lambda_{reply}}$$

It is extremely important to notice that $R_{Q_{server}}$ is *not* the under-load memory access latency, but it is the average time spent by a request at the server. However, to find out R_Q it is enough to take into account the base latency of the network as in 5.1.

$$R_Q = T_{req} + R_{Q_{server}} + T_{resp} \tag{5.1}$$

Results To evaluate the accuracy of the PEPA client-server model, we have done a test with the following scenario:

- the number of clients is fixed to $p = 16$.
- the average server service time is $T_S = 29\tau$. This value is typical of DRAM2 memories, as we have already mentioned in Section 4.3.2. We assume it exponentially distributed.
- the average *think* period T_P represents the degree of freedom. The distribution of the period is exponential. T_P will take its value in the range $[100\tau - 3000\tau]$. Being p fixed, it is necessary to vary T_P in a such a way to emulate all possible load states of the server, e.g unloaded, congested, partially congested and so on. As already told, since p is fixed to 16, for T_P values greater than 800τ the server is unloaded so cases of interest are in the range $[100\tau - 800\tau]$ that, moreover, is the typical range of T_P values that processes exploit in computational phases over a shared memory architecture.
- T_{req} and T_{resp} are evaluating on the Tilera Tile64 following the methodology explained in Section 2.2.1. The overall base memory access latency t_{a_0} is equal to 72τ as already reported in 4.3.2.

The under-load memory access time found through PEPA has been compared with the result of the following techniques:

- simulation performed with the JMT [4] Queuing Networks simulator, by means of which it has been implemented a client-server system.

- analytical resolution of the client-server model reported in 4.1. The comparison against more sophisticated resolution techniques or exploiting a server service time with a deterministic behaviour can be found in [15].

The graph in Figure 5.3 shows the progress of R_Q for T_P varying in the range $[0\tau - 3000\tau]$. It contains the shapes of all the technique, i.e. JMT simulation, PEPA, classical client-server model. The other two graphs (Figure 5.4) show respectively the absolute and relative error of analytical and numerical resolution techniques against the results retrieved by the simulation. The very important result is visible in Figure 5.4(b): the graph states that, for an exponential server, the PEPA approximation matches the simulation with a maximum relative error of 2%.

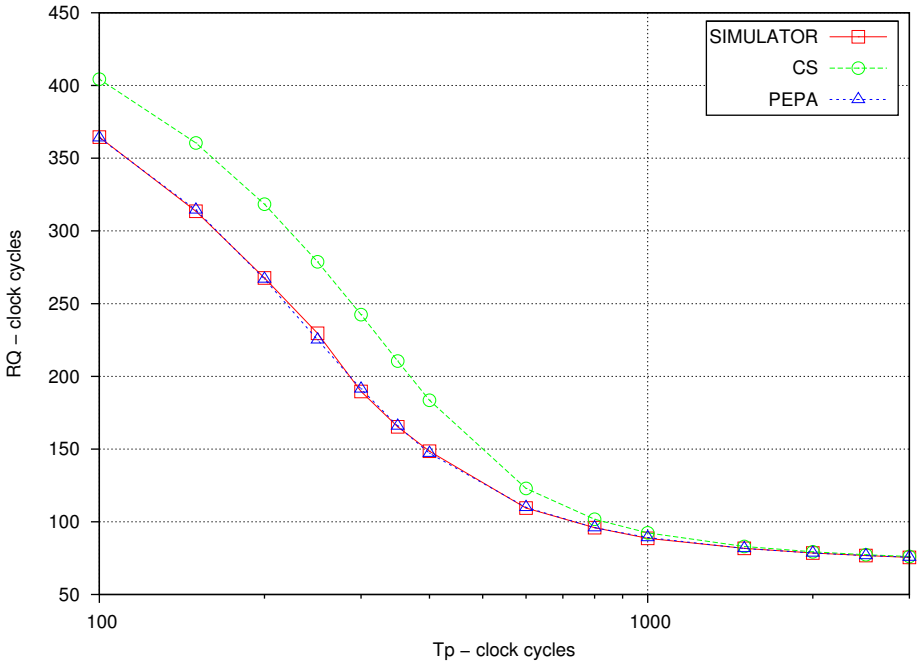
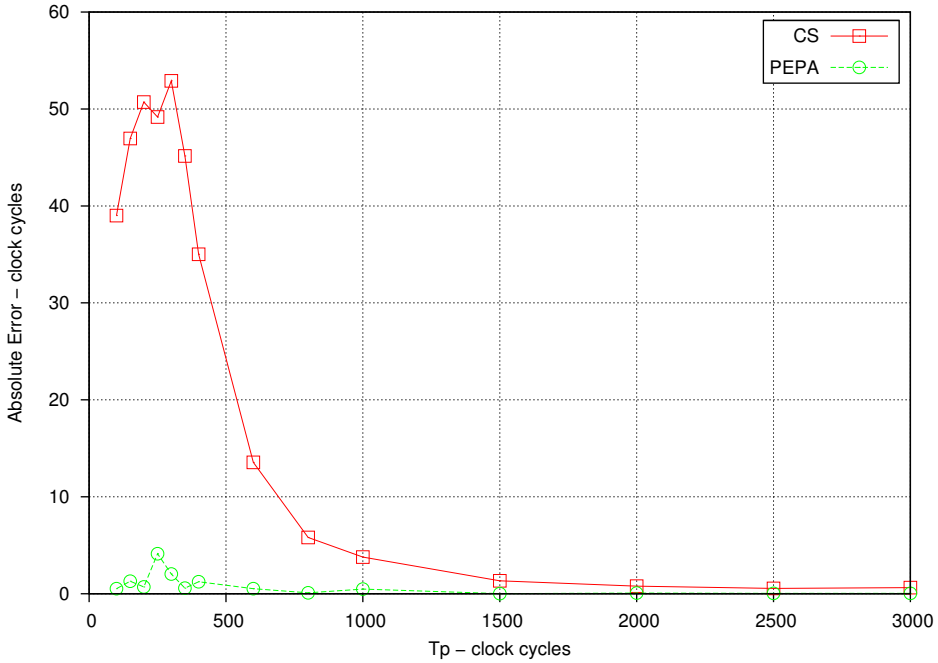
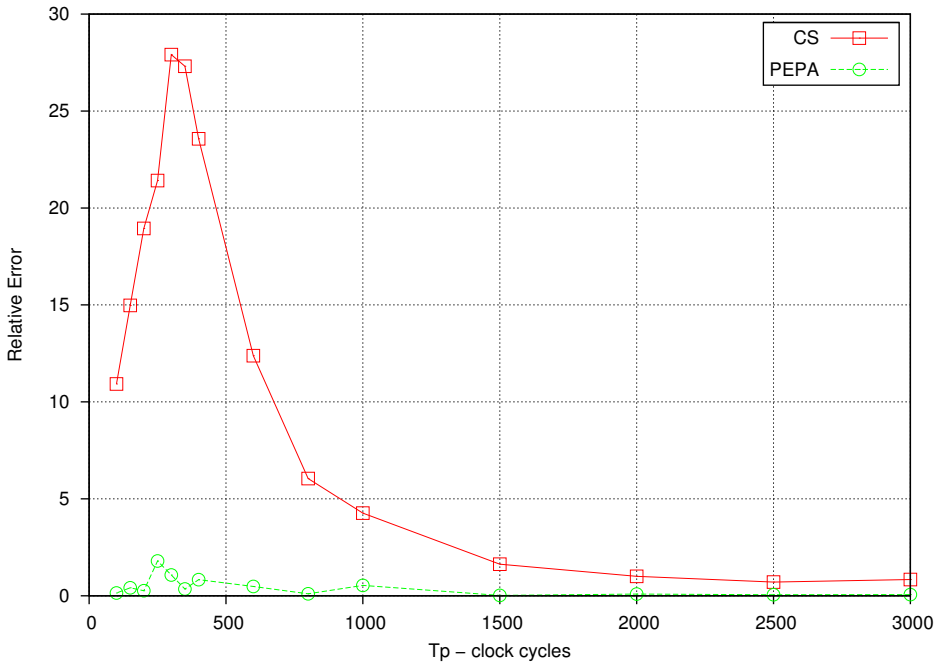


Figure 5.3: The RQ shapes



(a) Absolute Error



(b) Relative Error

Figure 5.4: Errors of resolution techniques against the simulation

5.3 Conclusion

Besides being an important step toward the modelling of complex parallel application-architecture systems utilizing a compositional and structured high-level approach, PEPA turns out to be useful even for what concerns the quality of the approximation because numerically resolution techniques are involved. In spite of this, the complexity of these resolutions is mitigated by advanced techniques and use of proper tools. What we are going to verify in the next chapters is the possibility for this new formalism to extend the client-server model taking into account application constraints or more complex architectures, e.g. shared memory hierarchies.

Chapter 6

Advanced Cost Models: impact of the Parallel Application

In the last chapter we introduced PEPA, an high-level formalism to generate Markov chains in order to specify and to solve the client-server model in a different manner with respect to the classical analytical approach reported in 4.2.

In this chapter we want to study if (and how) the impact of parallel applications could be modelled utilizing both analytical and numerical resolution techniques of the client-server model. We recall that parallel application are realized composing parallel paradigms in a structured approach. This way to operate will be fundamental for various assumptions that we will made during this chapter. The further intent is to verify what is the price to pay in terms of accuracy for these model enhancements so comparisons among techniques and against simulation will be made.

In particular, we want to remove some assumptions that we made in Section 4.2.2:

- processes in a parallel application could be different. An advanced cost model should treat this topic in order to be (more) precise.
- processes could have a complex internal behaviour that may affect the under-load memory access latency. The idea is that drastic phase-dependent changes in accessing the memory may change the congestion on the memory macro-module in a heavy way (the so called *bursts*).

The chapter has the following structure:

1. firstly, we give some definitions that will help us to formally recognize different processes or different phases of a process. Successively, we will see how this theory will be further applied in order to reduce the complexity of numerical resolution techniques in an orthogonal way with respect to techniques mentioned in 5.2.2.

2. secondly, we will study how is possible to catch the impact of process phases. Results of analytical and numerical resolution techniques will be compared against the output of a simulator developed in the University of Pisa.

3. finally, we will see how to model in PEPA a parallel application composed by different processes. Consequently, the obtained results will be compared against the JMT simulation and the analytical resolution technique introduced in 4.3.

6.1 Processes Classes and Processes Phases

Our intent is to model the workload given by a parallel application executing on a shared memory architecture. We want to do it because we claim that the under-load memory access latency R_Q could change a lot for different applications.

A first way to take into account the impact of a parallel application is to deal

- with heterogeneous processes, i.e. processes differ for own memory requests frequency or, equivalently, their T_P

- with a complex behaviour that a process could internally shows, i.e. computational phases followed by communication

In the previous chapters, we have already told about these aspects in an informal way. In order to give formal definitions for future treatments, we need to find a common point to decide when processes or phases differ. From the client-server model point of view, processes are modelled through modules that have a certain service time, that is the mean time between two consecutive memory requests. It is important to recall that this model

is based on mean value quantities rather than probability density functions making the analysis simpler and sufficiently accurate for our purposes. Considering that, the common point for discerning among processes or phases should be just what we have called T_P. In the following subsections, we will give definitions of *class of processes* and *process phase* in such a way will be possible to use them for future treatments.

6.1.1 Classes of Processes

The main idea is that processes belonging to the same class can be modelled in the same way, i.e. as homogeneous clients. Instead, processes of different classes should be modelled as heterogeneous clients. There are two main reasons for this classification:

- to establish in a formal way when a process differs from another one in such a way we will model them in a different way
- to recognize sets of homogeneous processes that will be modelled in a way to reduce the model resolution complexity, e.g. by mean of *aggregate* definition of clients

We have already mentioned that from the cost model point of view, the best parameter to classify processes is their T_P. However, this means to analyse all processes statically in order to derive their own T_P. In order to keep low the complexity of this procedure, we can exploit the benefits due to a structured parallel approach. Using always the same set of parallel paradigms to compose even complicated applications, the complier has to reason always on them. This means that it is able to immediately recognize sets of homogeneous processes inside a parallel application. For instance, consider a farm. The process classification can be done looking at the classical structure of the farm so all workers will belong to the same class because they are *replicated* processes, the emitter will belong to another class and the collector to another one. The same holds for all the other paradigms used in a structured approach.

In this way we are able to classify processes looking at the parallel application structure. This also means that it does not matter how a process is

made internally because we can discern among processes of a parallel application only looking at the used paradigm. On a hand, this way to operate reduces the complexity to analyse processes but, on the other hand, the processes classification is made on a base that is not the same of the client-server model, i.e. it does not take into account the fundamental parameter T_P. In fact, it may happen that two processes with same T_P fill a different role in a parallel application so they will belong to different classes. Apparently, this seems a problem but, again, following a structured approach to parallel programming, situations like this are rare and if happen, their impact on the resolution can be consider negligible.

It is worthwhile to notice that how to determine the T_P parameter is a different topic that we will treat in depth in the following. It is important to recall that up to now we were considering only processes characterized by an unique T_P easily determined by profiling. Of course, this does not hold in all cases and we will see why.

At this point, we have the way to recognize when processes differ in a non expensive way in order to model them as heterogeneous clients or no. Apparently, this classification does not seem to introduce other particular advantages. In fact, if a process is modelled as a client with a certain service time, i.e. the T_P, two processes with same T_P will be modelled as equal clients independently from their class. To note the further advantage we have to focus on the formalism wherein clients are represented in the various resolution techniques.

For instance, consider how clients are represented in PEPA. In general, each client is a component with own definition. In case we are able to recognize in somehow when processes are equal, we could decrease the number of that definitions with a potential benefit in terms of resolution. Suppose to have classified n processes in the same class C in this way:

$$C = \{p_1, \cdots, p_n\}$$

We know that processes in the same class should be modelled in the same way, so we can define an *aggregate* client component P that holds for all the n processes instead of having n distinct (but equal) definitions. This way to operate decreases the size and the complexity of the generated Markov

chain with global benefits in terms of resolution. As already mentioned, this approach allows to reduce the *state-space explosion* in an orthogonal way with respect to the techniques introduced in Section 5.2.2. We will see how to apply this theory in the next section when we will talk about the client-server model with heterogeneous clients.

6.1.2 Process Phases

Up to now we were considering processes executing only a computational phase characterized by a certain T_P. This way to model does not reflect properly the behaviour of processes. Structured parallel application have the further property to be composed by processes that alternate computational phases to communication. A very common example is the process starting with a computational phase (the so called *think* period) that will be followed by an inter-process communication (for instance a *send*). As already said, the difference between T_P in computational phases and T_P in communication phases could be even an order of magnitude.

Our main intent is to understand how phases may impact in a T_P derivation. A way to found the T_P parameter of a process is by inspection of the sequential code. We have:

$$T_P = \frac{T_c}{f} \tag{6.1}$$

where

- T_c is the completion time of the *sequential* version. It takes into account all the base latencies, e.g. interconnection structures, and various stall times, e.g. bubbles in the CPU pipeline
- f is the number of faults of the last memory support *exclusively private* of a processing node, i.e. that immediately before the shared memory hierarchy

It is important to notice that this technique is valid in case a process is (practically) always working. In fact, if the efficiency tends to one there are not *stopping* periods. Of course, in parallel applications processes are not always working. It may happen that the emitter of a farm or processes

implementing a *reduce* in a map-reduce do not have an efficiency that tends to one. According to the definition of a module efficiency in [20], we have:

$$\xi = \frac{T_{S_{id}}}{T_S} \tag{6.2}$$

where:

- $T_{S_{id}}$ is the *ideal* service time of the module. For simplicity, if we suppose to deal with the emitter of a farm, then its ideal service time is just the time to execute a send (we can consider negligible the rest of its behaviour, e.g. to receive and update the state of workers in case the farm is operating on demand).

- T_S is the *effective* service time of the module. We know that it can be found as

$$T_s = max\{T_A,\ T_{s_{id}}\}$$

At this point, the efficiency of the emitter can be rewritten as:

$$\xi = \frac{T_{send}}{T_A}$$

According to the theory, we would want that that the emitter and the workers are not bottlenecks in such a way to satisfy all requests (each one arriving every T_A).

So considering an optimal parallelism degree and a proper design of the emitter we have:

$$\frac{T_{send}}{T_A} < 1 \iff T_{send} < T_A$$

The direct consequence of this design is that in average the emitter interleaves *send* primitives to *stopping* periods wherein it waits for new incoming requests. In the latter period any code is executed so the mean time between two consecutive requests can not be derived only looking at the code of the *send* but should also be taken into account the stopping period. These situations recur frequently in structured parallel applications and stimulate solutions where phases are considerate in an explicit way. For this reason, we will give a first attempt to model processes not always working after the following treatments about phases.

6.2 How to deal with more Phases

As already mentioned, problems about the derivation of T_P can arise if processes show periods with sudden changes in memory requests frequency (the so called bursts). In case a computational phase with a certain T_{P_1} is followed by another one characterized by a T_{P_2}, the overall completion time T_c can be rewritten as the sum of the completion times T_{c_1} and T_{c_2} of the two phases. Reverting the Formula 6.1 we have that

$$T_{c_1} = f_1 \cdot T_{P_1}$$

$$T_{c_2} = f_2 \cdot T_{P_2}$$

where $f_1 + f_2 = f$. At the end, the overall T_P will be a weighted average based on the number of faults per phase:

$$T_P = \frac{T_c}{f} = \frac{f_1 T_{P_1} + f_2 T_{P_2}}{f_1 + f_2} \tag{6.3}$$

Being T_P a value expressed in clock cycles, we just take the integer part to be correct from a logical point of view.

Up to our studies, this was the best T_P evaluation also in case more phases are involved. The accuracy of all resolution techniques presented in previous chapters is worse if processes exploit more phases. Before showing the results and our efforts in order to improve the accuracy in somehow, we want to clarify the concept of process phase that we have applied in this context.

As already said in an informal way, a phase of a process is a lapse of time characterized by a certain T_P. At first sight, we can think that exist many levels of detail about to determine the beginning and the end of a phase. For instance, a phase could be the entire process life cycle or just the time between two consecutive memory requests. At this point, it is important to focus that we want to take into account phases in order to be more precise, but a complete and detailed treatment is not necessary because we recall that

1. our cost model is based on average values

2. significant variations in T_P values can be recognized only between computational and communication phases.

Considering that, we need again to find a way to establish phase boundaries. A compiler can easily find them concentrating on the base of some higher level aspects, e.g. well know software limits. For instance, every time a process invokes a *send* primitive, we can consider a phase the lapse of time needed to execute it.

Definition 5 *A process phase is a lapse of time characterized by a certain mean time between two consecutive memory requests and well recognizable software boundaries*

The above definition puts some constraints but does not cover all the cases. For instance, it may happen that T_P changes inside a computational phases, for instance in case a certain function is followed by another one quite different. In spite of this, we remark that substantial differences among T_P of computational phases are not present so the achieved level of details is sufficient for a good starting point.

6.3 Process Phases Modelling

Having the theory to recognize process phases, we can evaluated their impact on the model. In this section we report:

- firstly, a brief summary on the effectuated tests about the impact of process phases modelled according to the Formula 6.3. It is worthwhile to recall that in this solution the T_P derivation is made by mean of weighted average value based on number of faults.
- successively, we present different ways to deal with process phases. For each solution we will show results and comparisons with previous versions.

Before doing that, we want to explain the test case. First of all, we have reproduced by simulation the behaviour of processes exploiting two different phases, i.e. a computational one (called *think*) followed by an inter-process

communication (*send*), on a shared memory architecture. We have done it using an architectures simulator developed by our research group in the University of Pisa.

The test case is the following:

- the number of processes in execution on the same amount of processing nodes is fixed to $p = 16$.
- the average memory macro-module service time is $T_S = 29\tau$. This value is typical of DRAM2 memories, as we have already mentioned in Section 4.3.2. We assume it exponentially distributed as usual.
- the mean time between two consecutive memory requests during the phase *think* (T_{P_t}) represents the degree of freedom and it will take values in the range $[200\tau - 800\tau]$. Instead, in the phase *send*, T_{P_s} is fixed to 20τ. We can notice that the difference between T_{P_t} and T_{P_s} is an order of magnitude.

6.3.1 Phases by mean of Weighted Average Value

We recall that the value of T_P in this technique is found according to the Definition 6.3. The under-load memory access time found by simulation has been compared with

- the one found by the classical analytical resolution of the client-server model reported in 4.2
- the one obtained with the numerical resolution via PEPA explained in 5.2.1

Figure 6.1 shows all the RQ shapes, i.e. SIMULATOR, CS and PEPA-WA (WA stays for weighted average) while the remainder show respectively the absolute and relative error.

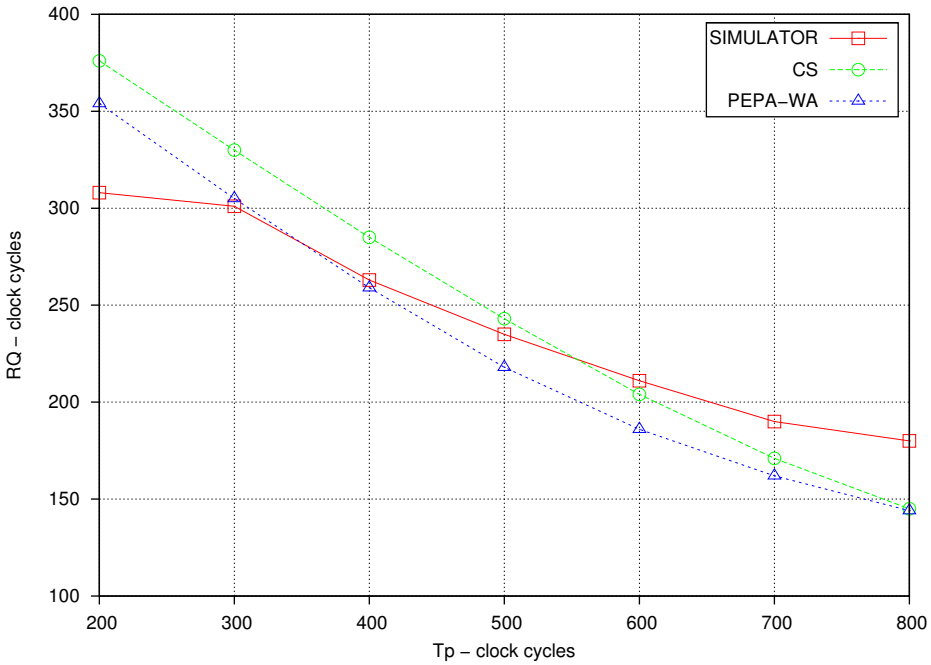
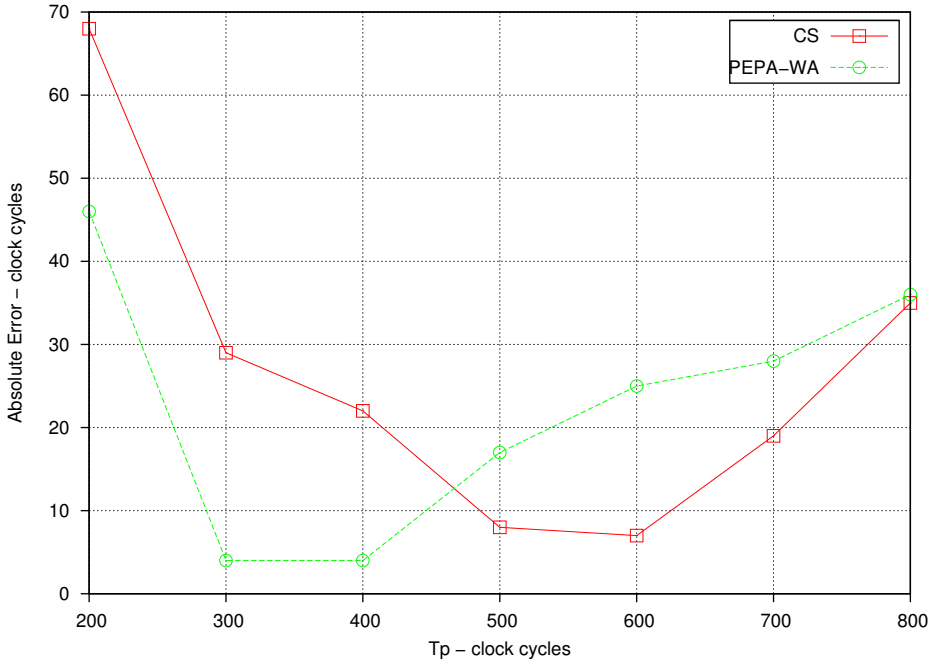
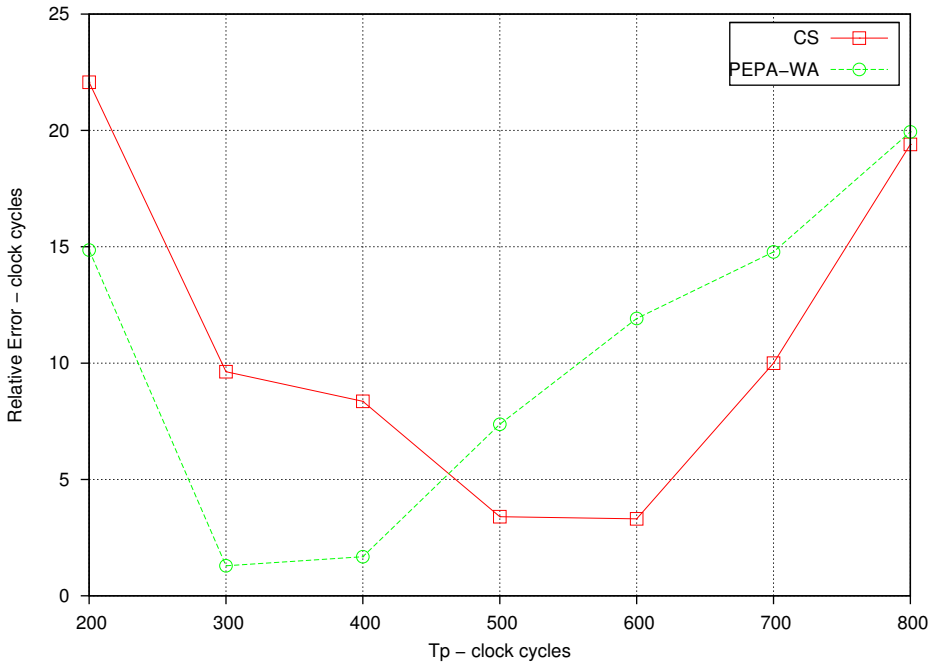


Figure 6.1: Under-load Memory Access Latency.



(a) Absolute Error.



(b) Relative Error.

Figure 6.2: Errors with respect to the simulation.

Comments First of all, we notice that for lower T_P the numerical resolution is better than the analytical one while it is the opposite for higher T_P. Both resolutions show a maximum relative error around the $10\% - 15\%$ in the range $T_P = [300\tau - 700\tau]$ while on extreme T_P values the relative error is bigger. Anyway, we would want it lower in all the range.

We tried to model the phase impact in other ways in order to reduce the gap between the under-load memory access latency of at least one resolution technique and the simulation. In the next subsections we will present these techniques.

6.3.2 Explicit Phases

The first idea is to model process phases in an explicit way as shown in Figure 6.3. This is exactly what happen to processes: different phases, each one effectuating memory requests with own rate, interleave among them. The idea is to catch the frequency wherein a process passes from a generic phase to another one.

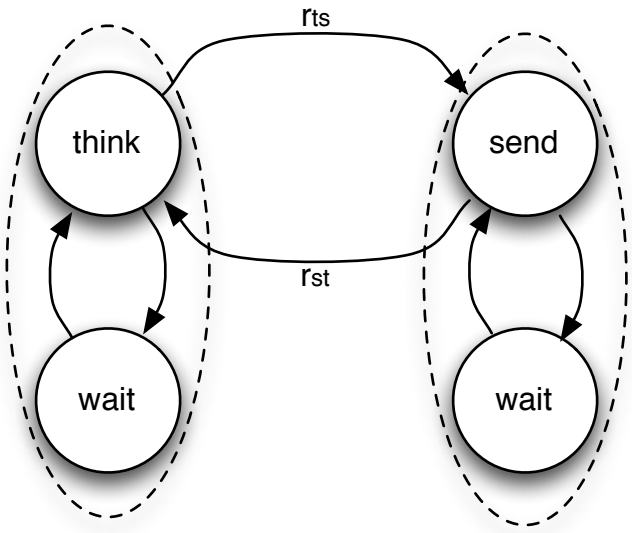


Figure 6.3: The behaviour of a client exploiting a *think* phase followed by a *send* one.

An example on how this can be achieved in PEPA is reported below. Basically, we change the definition of client in such a way will be also possible to pass from a phase to another one.

$$\begin{array}{rcl} C_{think} & \stackrel{def}{=} & (request, r_{request_t}).C_{wait_t} + (send, r_{ts}).C_{send} \\ C_{send} & \stackrel{def}{=} & (request, r_{request_s}).C_{wait_s} + (think, r_{st}).C_{think} \\ C_{wait_t} & \stackrel{def}{=} & (reply, \top).C_{think} \\ C_{wait_s} & \stackrel{def}{=} & (reply, \top).C_{send} \end{array}$$

Assume the two phases involved in Figure above, then frequencies r_{ts} and r_{st} are easily found reverting their period:

$$r_{ts} = \frac{1}{f_t \cdot T_{P_t}}$$

$$r_{st} = \frac{1}{f_s \cdot T_{P_s}} \tag{6.4}$$

Apparently, this way to operate has a problem. As already explained above, we are effectuating estimations on the base of the sequential code inspection so the length of phase periods are evaluated only considering the number f of faults and the mean time between two consecutive memory requests (T_P). The length of a phase is crucial in our treatment because it influences directly the rate of some actions. Of course, problems arise when more processes are in execution because the impact of the under-load memory access time.

Consequently, the rate estimations 6.4, that are found statically, can differ a lot from the effective ones. So phase periods should not be evaluated only considering the sequential version, i.e. $f \cdot T_P$, but have to include the impact of R_Q too. This can be done in various ways, a solution is to adopt an iterative approach. However, we have to keep in mind that we want to keep low the complexity to found R_Q, so we prefer to introduce an approximation respect to complex procedures. Therefore, the length of phases can be estimated using the *base* memory access latency t_{a0} that can be derived easily at compilation time. Considering that, we set the rates in the following way:

$$r_{ts} = \frac{1}{f_t \cdot (T_{P_t} + t_{a0})}$$

$$r_{st} = \frac{1}{f_s \cdot (T_{P_s} + t_{a0})} \tag{6.5}$$

As already explained in Section 2.3.2, basically t_{a0} is the sum of terms: the latency L_s of the server in case no conflicts are taken into account and the base network latency $T_{net} = T_{req} + T_{resp}$:

$$t_{a0} = L_s + T_{net}$$

The idea is to substitute the base latency L_s of the server with the under-load memory access latency $R_{Q_{server}}$ found with PEPA in order to obtain no more t_{a0} but R_Q:

$$R_Q = R_{Q_{server}} + T_{net}$$

Results and comments Figure 6.4 shows the under-load memory access latency R_Q of the simulation (SIMULATOR), the previous version (PEPA-WA) and the new version with explicit phases (PEPA-EP). Figure 6.5 shows the absolute and relative error.

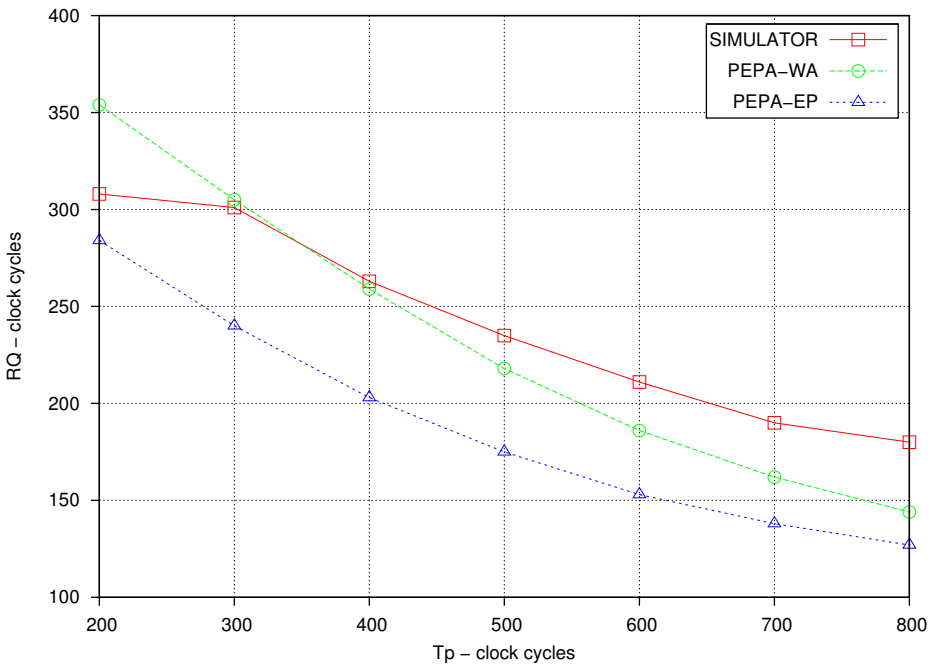
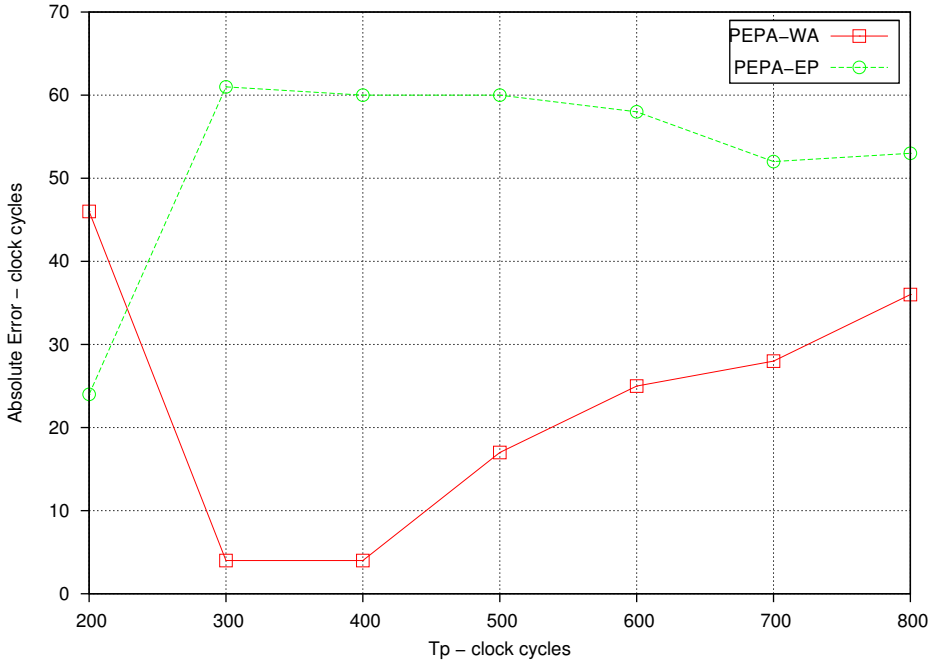
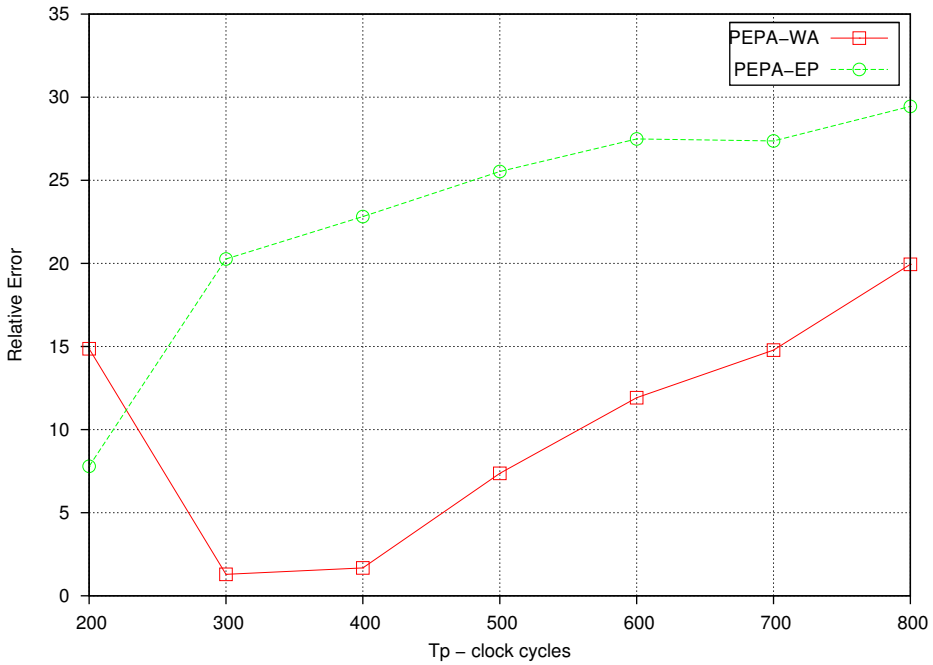


Figure 6.4: Under-load Memory Access Latency.



(a) Absolute Error



(b) Relative Error

Figure 6.5: Errors against the simulation of both numerical resolutions

We can see that the relative error (we call it ϕ) of the new version (PEPA-EP) is higher than the error of the previous version (PEPA-WA). This is due to the approximation that we introduce in the evaluation of phases periods. Considering that, we can also affirm that the error depends by the difference between t_{a0} and R_Q hence it will be greater if their ratio grows:

$$\frac{R_Q}{t_{a0}} \uparrow \Longrightarrow \phi \uparrow$$

However, the error in this new version is almost constant in all the range and no more with enormous changes as before.

The potential advantage of this technique is that we believe that it could be used in order to model processes not always working. In fact, it is sufficient to pass from the *think* phase to a *stopping* one where no memory requests are generated. It is important to notice that the accuracy still depends by the ratio $\frac{R_Q}{t_{a0}}$.

How this can be achieved in PEPA is shown in the following code:

$$\begin{array}{rcl} C_{think} & \stackrel{def}{=} & (request, r_{request_t}).C_{wait_t} + (stop, r_{ts}).C_{stop} \\ C_{stop} & \stackrel{def}{=} & (think, r_{st}).C_{think} \\ C_{wait_t} & \stackrel{def}{=} & (reply, \top).C_{think} \end{array}$$

First of all, we recall that processes with efficiency $\xi < 1$ are not bottlenecks so their effective service time T_S is equal to the inter-arrival time T_A. Thanks to the structured parallel approach, we are able to estimate the effective service time of a process looking at T_A that is a well know input parameter. Considering that, the rates to switch from a generic phase to another one should be setted in this way:

$$r_{ts} = \frac{1}{T_{s_{id}}} = \frac{1}{f_t \cdot (T_{P_t} + t_{a0})}$$

$$r_{st} = \frac{1}{T_S - T_{s_{id}}} = \frac{1}{T_A - T_{s_{id}}}$$

If we apply the formulas above to the example of the Section 6.1.2, we have that the emitter states the following parameters:

$$T_{s_{id}} \simeq T_{send} \Longrightarrow \begin{cases} r_{ts} = \frac{1}{T_{send}} \\ r_{st} = \frac{1}{T_A - T_{send}} \end{cases}$$

Unfortunately, this scenario has not been simulated yet so we have no results about this hypothesis, but it belongs certainly to future works.

6.3.3 Phases by means of Average Clients

Another way to deal with more phases is to estimate how many processes are in average in a certain phase. Consider the test case introduced in Section 6.3 with a number of processes equals to p. If we are able to recognize that p_t processes are in average in the *think* phase and p_s processes are in *send* phase, we could instantiate the client-server model with heterogeneous clients (see Section 6.4) where n clients have T_{P_t} as service time while the other m have T_{P_s}.

The problem lies in how to evaluate the average number of clients in a certain phase. The way to do it is to consider again the length of phases in such a way to found their ratio r. Successively r can be used in order to evaluate p_t and p_s as

$$p_s + r \cdot p_s = p \Longrightarrow p_s = \frac{p}{1+r}$$
$$p_t = r \cdot p_s$$

The problem is again to find the right value of r because it depends by the length of phases. The solution is to consider the base memory access latency t_{a0} another time:

$$r = \frac{f_t \cdot (T_{P_t} + t_{a0})}{f_s \cdot (T_{P_s} + t_{a0})}$$

It is important to say that this way to operate introduces an ulterior approximation. In fact, being p_s and p_t an average number of processes either in *send* or in *think* phase, they must be integer numbers in order to use them in the client-server model with heterogeneous clients. Therefore, we need to approximate the found values to the closer integers. Of course, this approximation worsens the accuracy of the technique with respect to the previous one.

6.3.4 Explicit Phases with Average Clients

This solution is a mix of the previous versions. The idea is to use explicit phases in order to determine the average number p_i of clients in a certain phase i. Once we have found these values, we calculate R_Q as weighted average value on the number of clients p:

$$R_Q = \frac{\sum_i R_{Q_i} \cdot p_i}{p} \tag{6.6}$$

Each phase i has own R_{Q_i} that, as such, can be evaluated as usual solving the client-server model only considering that phase. At this point, we know all the R_{Q_i} and we have to evaluated the average number p_i of clients for each phase i. We recall that the frequency to pass from a generic phase i to another phase j depends by R_{Q_i} and that, following the approach explained in the Explicit Phases technique, phases can be modelled as explicit ones that interleave among them. If we are able to model all phases with the associated frequencies to pass from one to another one, the number of clients in a certain phase is given by the population of that phase in steady-state condition of the system.

We try to clarify the way to operate with a simple example. We consider only two phases: *think* and *send*. The former has a certain T_{p_s} while the latter is characterized by T_{p_s}. First of all, we apply the classical client-server model in order to derive respectively R_{Q_t} and R_{Q_s}.

At this point we use R_{Q_t} and R_{Q_s} *as input* in a new system composed by p modules each one characterized by *think* and *send* states (Figure 6.6). Of course, frequencies between phases are not more influenced by t_{a0} as in Equations 6.5, but in this way:

$$r_{ts} = \frac{1}{f_t \cdot (T_{P_t} + R_{Q_t})}$$

$$r_{st} = \frac{1}{f_s \cdot (T_{P_s} + R_{Q_s})}$$

Once we have found the steady-state solution of the new system, we have that the population in the state *think* is the average number p_t of processes in that state. The same holds for the state *send*. At this point we rewrite the Equation 6.6 for this specific case founding R_Q:

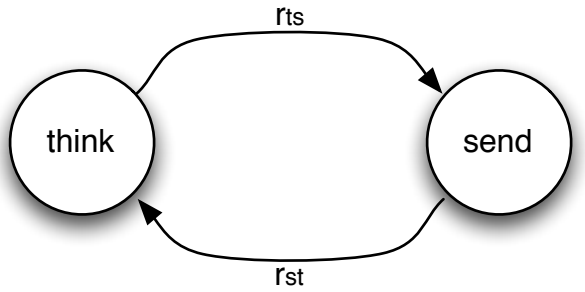


Figure 6.6: Explicit behaviour of a process.

$$R_Q = \frac{R_{Q_t} \cdot p_t + R_{Q_s} \cdot p_s}{p_t + p_s}$$

This technique can be easily adopted with PEPA. We have already seen in Section 5.2.1 how to define a classical client-server model in this formalism. The code below simply states the new system composed by p components each one characterized by a *think-send* behaviour.

$$\begin{array}{rcl} C_{think} & \stackrel{def}{=} & (send, r_{ts}).C_{send} \\ C_{send} & \stackrel{def}{=} & (think, r_{st}).C_{think} \end{array}$$

$$C_{think}[p]$$

Results and comments Figure 6.7 shows the under-load memory access latency R_Q of the simulation (SIMULATOR), the versions PEPA-WA and PEPA-EP and the final version PEPA-EPAC. Figure 6.8 shows the absolute and relative error.

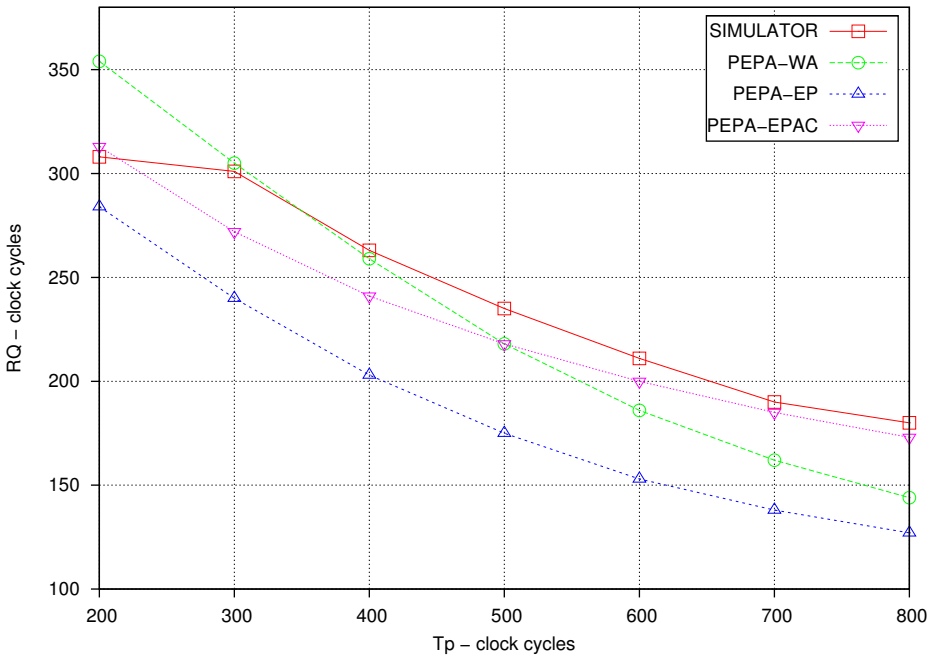
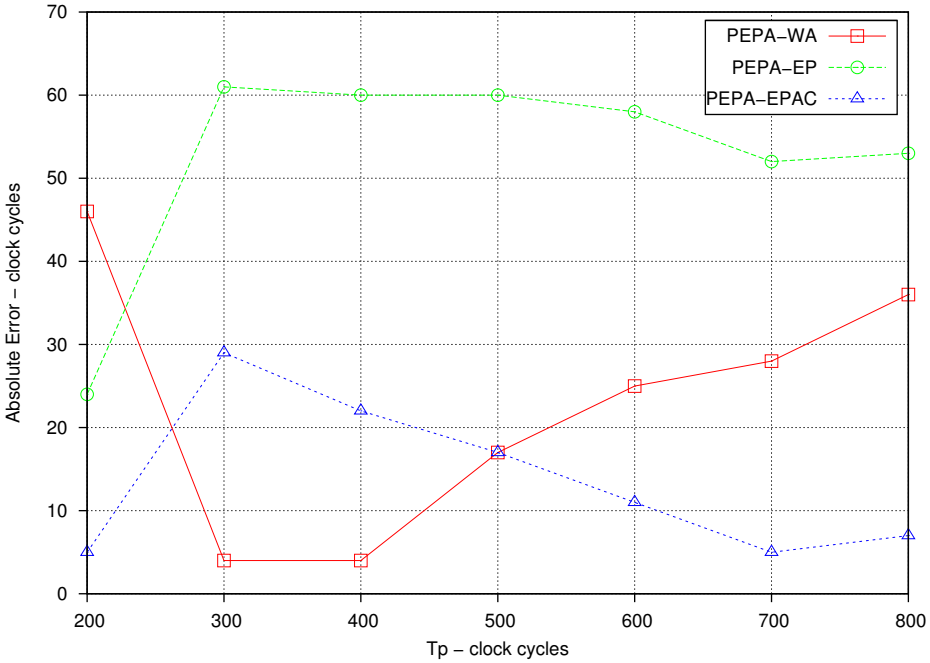
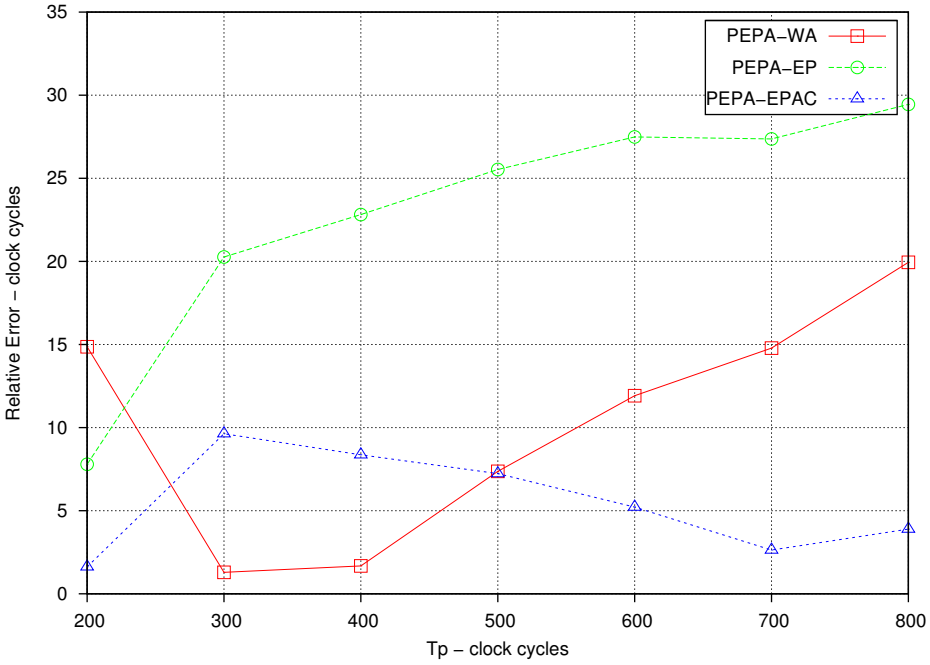


Figure 6.7: Under-load Memory Access Latency.



(a) Absolute Error



(b) Relative Error

Figure 6.8: Errors against the simulation of numerical resolutions

In general this final solution gives a general improvement in accuracy since the relative error never exceed the 10%. Further, it can also be utilized to model processes not always working. In fact, it is sufficient to evaluate R_{Q_t} and successively to set the rates as

$$r_{ts} = \frac{1}{T_{s_{id}}} = \frac{1}{f_t \cdot (T_{p_t} + R_{Q_t})}$$

$$r_{st} = \frac{1}{T_A - T_{s_{id}}}$$

Summarizing, this new version exhibit a better accuracy than PEPA-WA and succeed to model processes not always working as in PEPA-EP.

6.3.5 Comments

We have formalized the concept of processes class in order to be able to discern among processes without doubt and to introduce optimizations able to reduce the resolution complexity. Anyway, a first example on how this definition are applied is immediately shown in the next section.

We have seen that a process classification should be made taking into account the mean time between two consecutive memory requests, i.e. T_P, because this parameter is at the base of the client-server model, but this is not possible without to increase the analysis complexity. Therefore, process classification is made looking at the structure of the parallel application.

Successively, it has been explained how R_Q could also be derived in case processes show a complex internal behaviour, i.e. the so called phases. Therefore, a definition of process phase has been formalized and some techniques to deal with phases have been proposed and analysed.

6.4 Heterogeneous Clients in PEPA

In this section we will see how to model in PEPA a parallel application on a shared memory architecture and composed by processes with different T_P. We recall from Section 4.3 that an example could be the functional partitioning with independent workers. Of course, heterogeneous clients must be involved in order to model processes with different T_P.

6.4.1 Definition

The PEPA program taking into account heterogeneity is shown below. Thanks to the compositional approach of PEPA, we can directly reuse the same server component and the definition of a generic client already present in Section 5.2.1. So basically, a generic client has the same behaviour as before and this implies that is unnecessary to add further operations apart from the already used *request* and *reply*. As a consequence of this structured approach, also the cooperation set in the last expression remains the same.

Of course, a change occurs in the number of client definitions. In fact, we want to apply the theory seen above in order to recognize C classes of processes. According to the theory, we do not want to have a definition per client but, in order to keep lower the resolution complexity, there must a number of client definitions equals to C. Every definition has own rate of request, that is peculiar for that given class. This rate is the inverse of the T_P characterizing the class and it has been found according to the techniques explained in the previous section, i.e. by profiling in the easiest cases or using the explicit phases technique. The last expression of the program defines the overall system in which clients of C classes run in parallel synchronizing themselves with the server. Obviously, each class of clients specifies the number of clients belonging to that class.

$$
\begin{aligned}
Client_{1_{think}} &\stackrel{def}{=} (request, r_{request_1}).Client_{1_{wait}} \\
Client_{1_{wait}} &\stackrel{def}{=} (reply, \top).Client_{1_{think}} \\
&\ldots \\
Client_{C_{think}} &\stackrel{def}{=} (request, r_{request_C}).Client_{C_{wait}} \\
Client_{C_{wait}} &\stackrel{def}{=} (reply, \top).Client_{C_{think}} \\
Server &\stackrel{def}{=} (request, \top).Server + (reply, r_{reply}).Server
\end{aligned}
$$

$$
Client_{1_{think}}[p_1] \parallel \ldots \parallel Client_{C_{think}}[p_C] \underset{request,reply}{\bowtie} Server[1.0]
$$

6.4.2 Quantitative Comparison with respect to other Resolution Techniques

Model Resolution Following the procedure in 5.2.2, we have to found $R_{Q_{server}}$ in order to evaluate the under-load memory access latency. Having more *wait* states, i.e. one for client definition, we can evaluate the average number of clients staying in the state $Client_{wait}$ as

$$R_{Q_{server}} = \sum_{i=1}^{c} \frac{p_{wait_i}}{\lambda_{reply}} = \frac{\sum_{i=1}^{c} p_{wait_i}}{\lambda_{reply}} \tag{6.7}$$

where p_{wait_i} is the average number of clients belonging to the state $Client_{i_{wait}}$ in steady-state condition of the system. Successively, it is sufficient to add the base network latencies for the request and for the reply to obtain the under-load memory access latency as usual:

$$R_Q = T_{req} + R_{Q_{server}} + T_{resp} \tag{6.8}$$

Results We tested and compared the accuracy of this resolution technique against the results found for the test case defined in Section 4.3.2. We briefly report the features of the scenario:

- the number of clients is fixed to $p = 16$; seven of them have a certain service time T_{P_1}, other seven a T_{P_2} while the last two have a fixed $T_P = 100\tau$. The idea is to simulate a functional partitioning with independent workers and two service processes, i.e. a dispatcher and a gather. We recall that processes exploit only a phase in analogy to the example in Section 4.3.2.

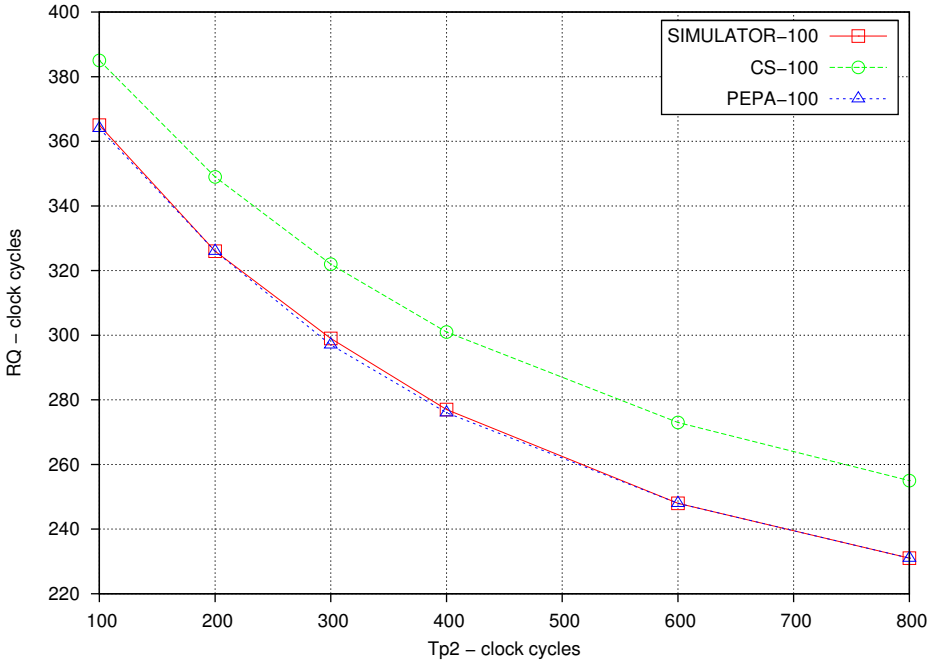
- the distribution is exponential for all the service times. Since p is fixed, the service times of the clients are the degree of freedom. In particular, in each test T_{P_1} will be fixed to a certain value chosen in the range $[100\tau - 800\tau]$ while T_{P_2} will vary in the same range in such a way will be possible to find results for different load states of the server.

The graphs in Figure 6.9 and 6.10 show the behaviour of R_Q varying T_{P_2} and with T_{P_1} fixed respectively to 100τ, 300τ and 500τ. In particular, the graphs state the R_Q shape of the JMT simulation (SIMULATOR), PEPA

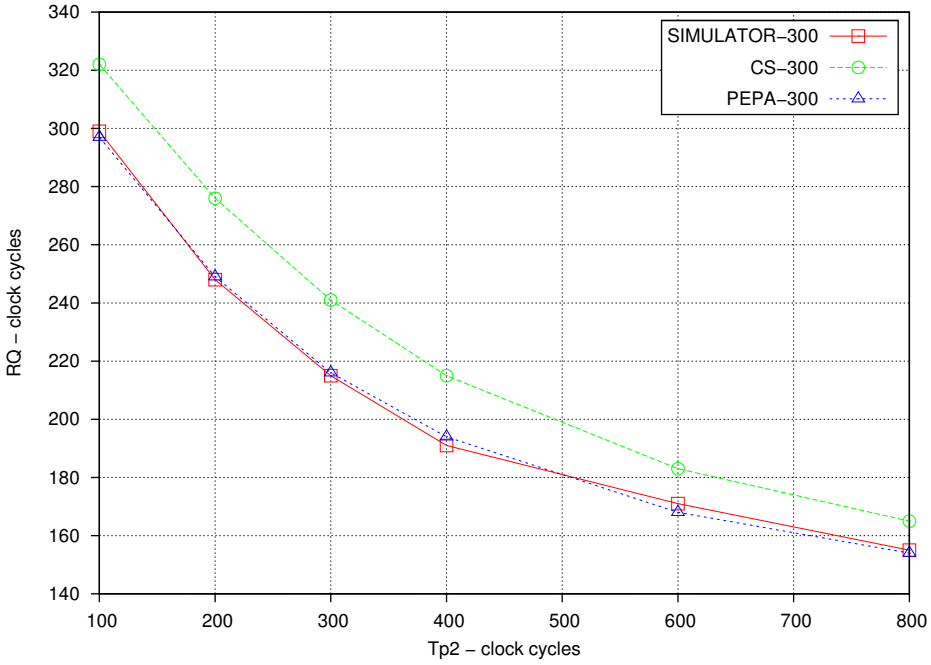
and the analytical resolution introduced in Section 4.3.2 (CS). Successively, absolute and relative errors of PEPA and CS approaches against the results of the simulation are shown and compared.

As already explained in previous comments (Section 4.3.2), CS resolution exhibits in general a gap in the R_Q shape with respect to the one of the simulation. This phenomenon is worsen in case heterogeneous clients are involved, especially if the service time of different clients differs a lot. On the other hand, we pointed out in previous chapter that the PEPA approach does not introduce errors when homogeneous clients are involved.

On the base of the results presented here, we can clearly conclude that the PEPA approach does not suffer in accuracy even introducing heterogeneous clients. In fact, the maximum relative error in all graphs is below than 2%, that is the threshold that we have found for homogeneous clients in PEPA.

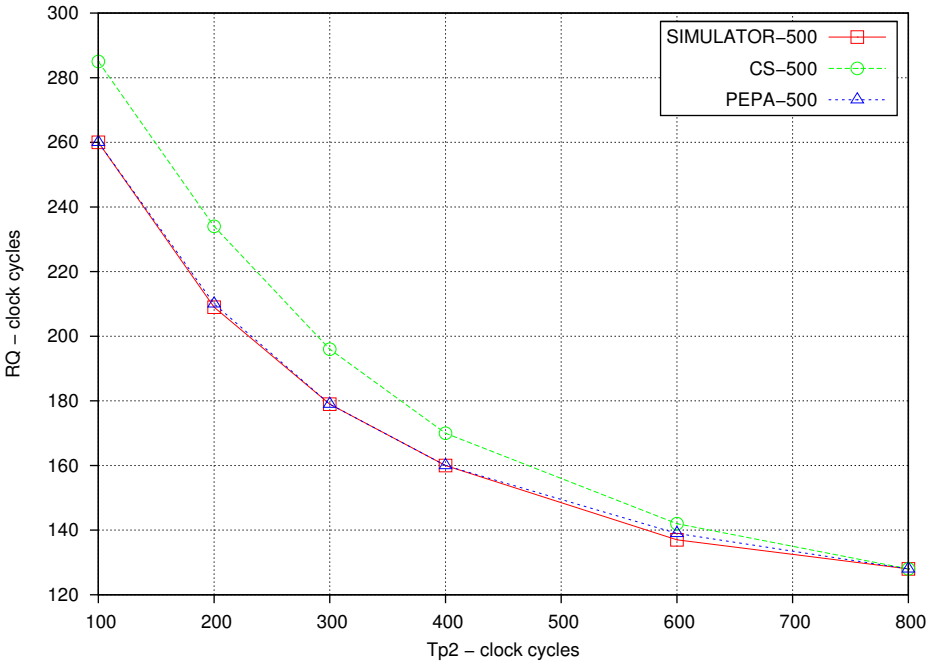


(a) $T_{P_1} = 100\tau$



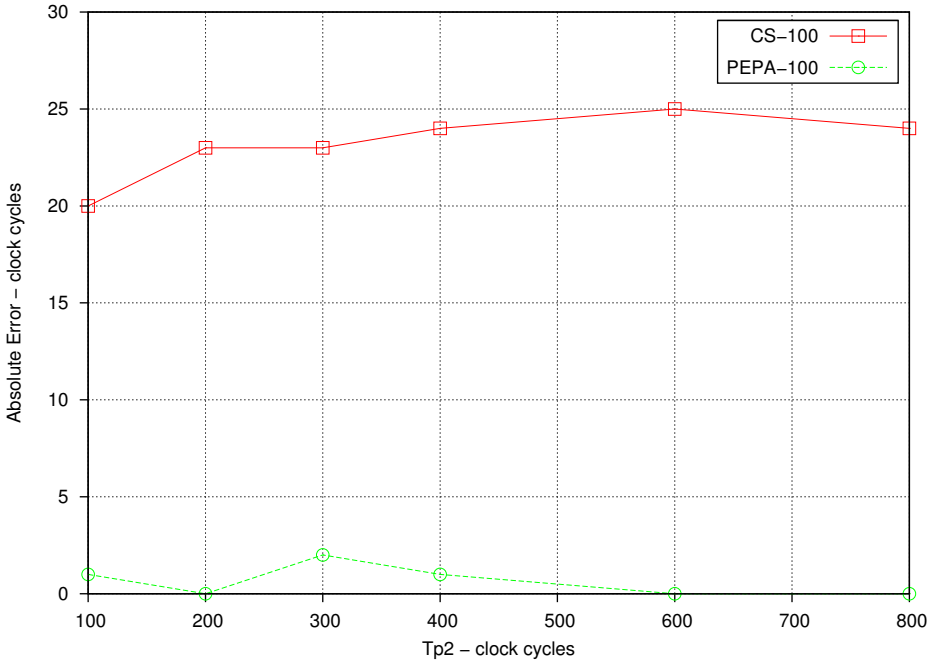
(b) $T_{P_1} = 300\tau$

Figure 6.9: Under-load Memory Access Latency.

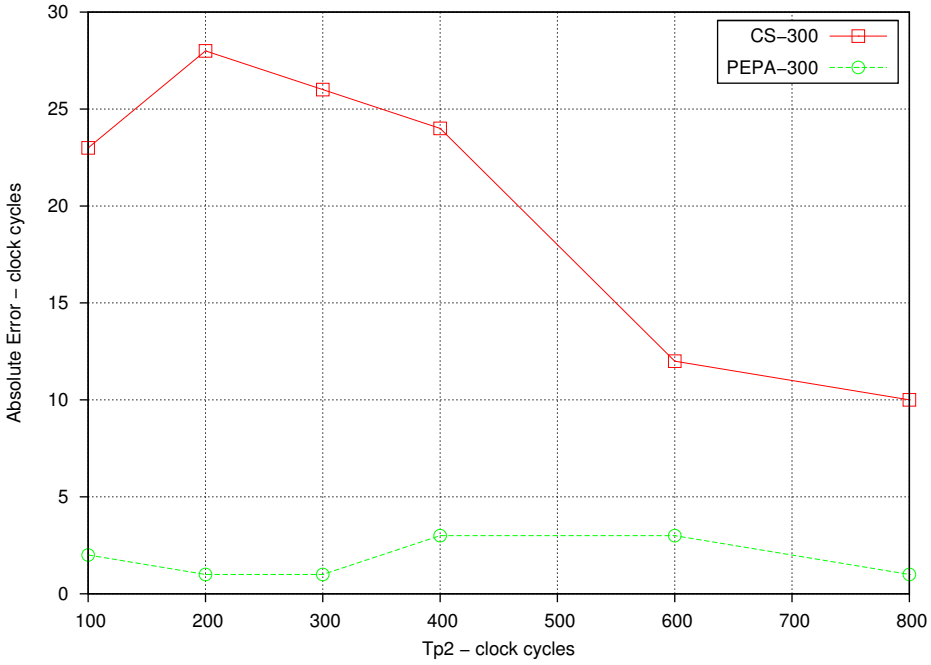


(a) $T_{P_1} = 500\tau$

Figure 6.10: Under-load Memory Access Latency.

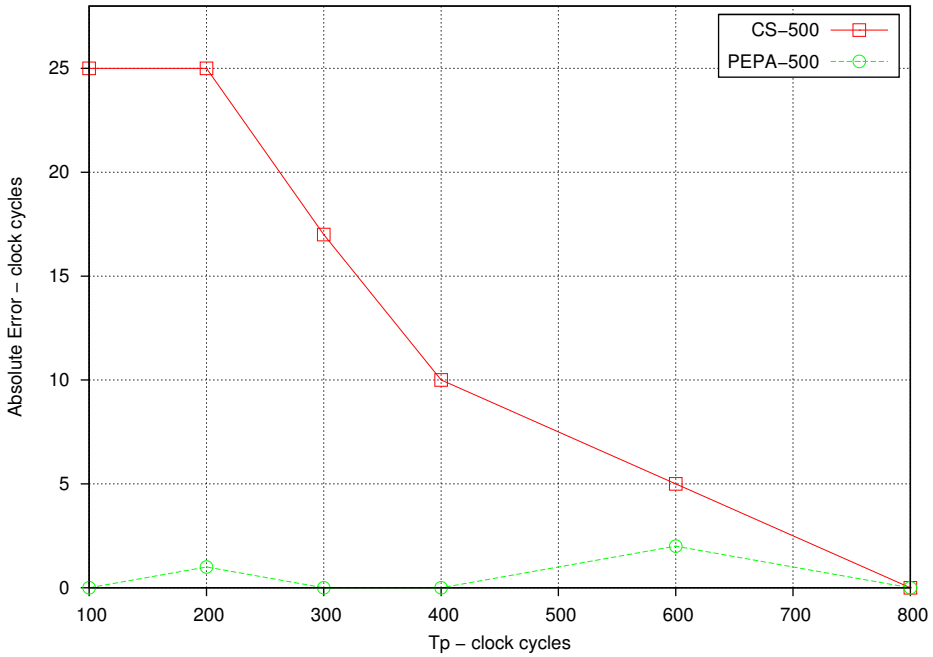


(a) $T_{P_1} = 100\tau$



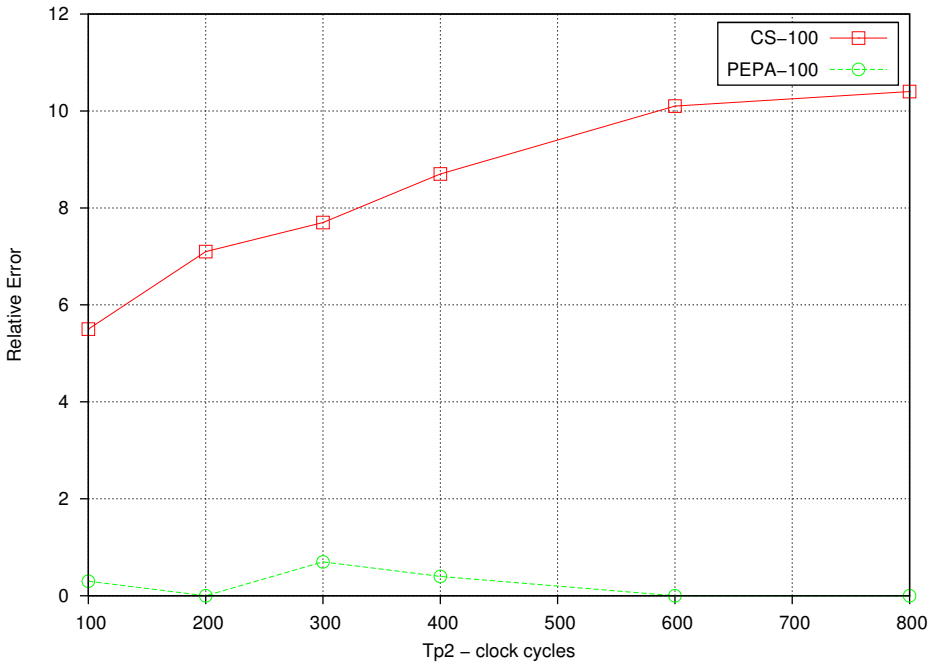
(b) $T_{P_1} = 300\tau$

Figure 6.11: Absolute Error.

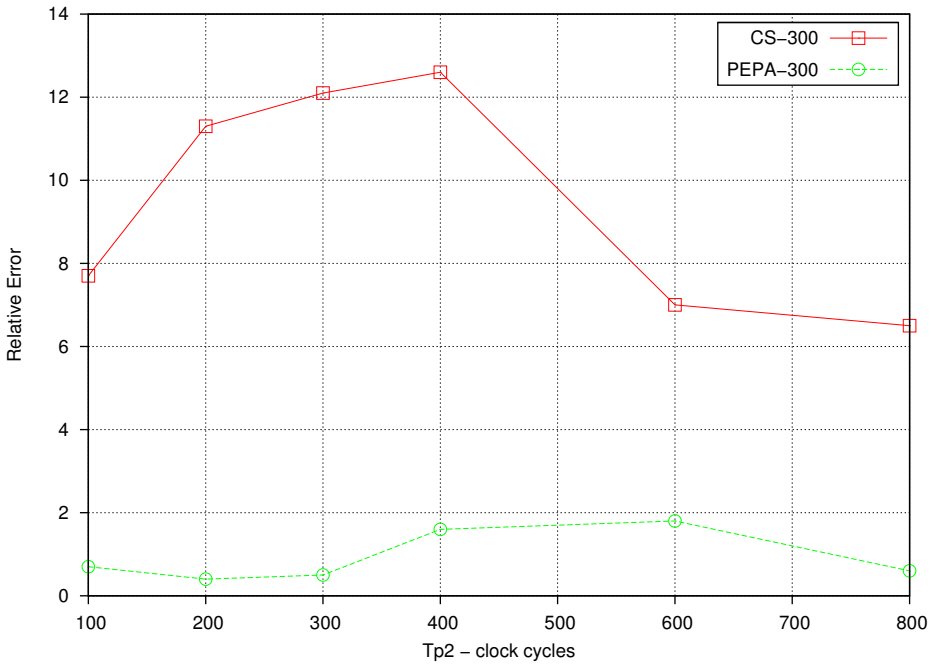


(a) $T_{P_1} = 500\tau$

Figure 6.12: Absolute Error.

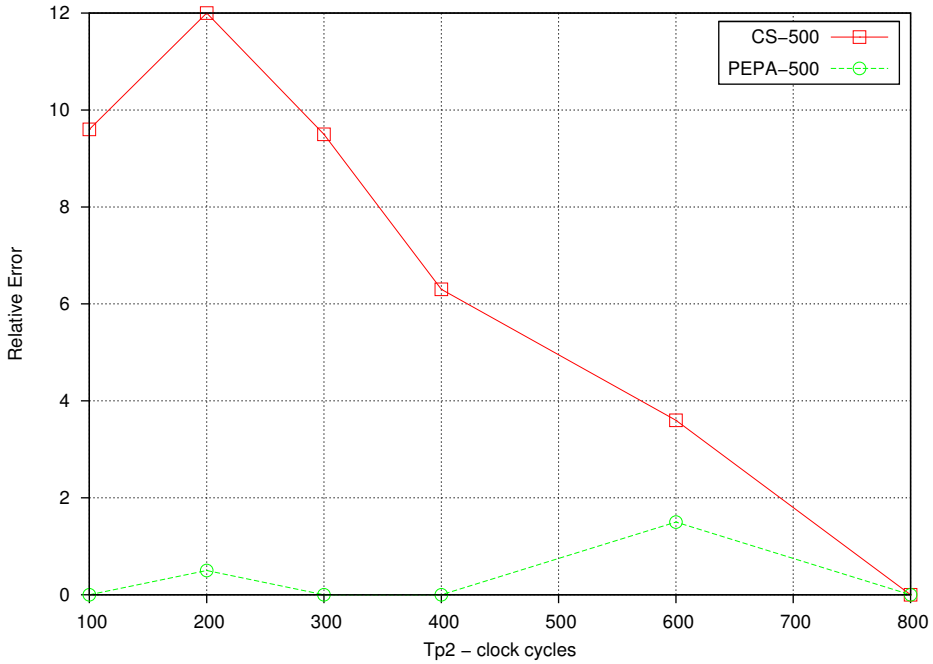


(a) $T_{P_1} = 100\tau$



(b) $T_{P_1} = 300\tau$

Figure 6.13: Relative Error.



(a) $T_{P_1} = 500\tau$

Figure 6.14: Relative Error.

6.5 Conclusion

On the basis of the client-server model there is the evaluation of the time between two consecutive memory requests generated by a process in execution of a processing node, i.e. the so called T_P. This derivation is a crucial point in order to obtain accuracy of the involved resolution techniques. In fact, we have seen that good T_P derivations bring results practically identical to the ones obtained by simulation at least for numerical resolution techniques.

In this sense, we can conclude that the major impact of parallel applications is due to phases that complicate the derivation of this fundamental parameter. Consequently, the accuracy worsens even if sophisticated techniques are utilized.

Chapter 7

Advanced Cost Models: Hierarchical Shared Memory

Hierarchical shared memory is peculiar to multi-cores, especially if the trend follows the direction that has been taken. In these architectures, more than one level of memory hierarchy is shared by processing nodes (see Section 2.1). Therefore, conflicts for accessing shared resources could become significant for what concerns the under-load memory access latency because more queues must be travelled. In fact, if we consider for instance that also the second level of cache is shared among all (or only a subset of) cores, we have that this memory will be a first level of queue while the main memory (that continues to be shared) will be the second one. This behaviour can be extended up to a general number of hierarchy levels, e.g. also the third level of cache could be shared and so on. Of course, memory requests will travel all the queues only in some cases.

Our main goal is to measure the impact of this hierarchy of queues in terms of performance indexes, i.e. the under-load memory access time R_Q. Of course, we want to do it according to the methodology followed so far hence the starting point will be the same, i.e. the client-server model.

7.1 Hierarchical Client-Server Model with Request-Reply Behaviour

Before starting to explain how to enhance the classical client-server model in order to deal with shared memory hierarchies, we do some initial assump-

tions:

- we consider a shared memory hierarchy composed by a second level of cache $L2$ and a main memory M
- M is shared among all the processing nodes. In the following, we concentrate only on a macro-module that we will consider shared among all the processing nodes. We will refer at it as M
- $L2$ is shared among *disjointed* subsets of processing nodes. This means that the same (and fixed) number n of processing nodes can access own $L2$
- the size of $L2$ is big enough to contain the entire working sets needed to processes to execute their job (obviously, we are considering only processes in execution on processing nodes that belong to the same subset, i.e. they share the same $L2$). In this way, we abstract the impact due to techniques for replacing blocks, i.e. the different ways to address caches

Briefly, we can summarize the behaviour of a parallel application executed on a shared memory architecture with the above assumptions in the following way.

A processing node P (executing a process), in case a cache fault occurs, generates a memory request req toward the second level of cache $L2$ that, as usual, will answer directly to the processing node in case the requested cache block belongs to it. Otherwise, req will be forwarded to M and $L2$ has to wait the reply from the memory before responding to the processing node. It is worthwhile to notice that in the mean time that $L2$ waits for a answer from M, it can reply to other requests of processing nodes provided that it is able.

In the former case, P has to wait the time needed for

- the request req to reach $L2$ travelling an interconnection network $P-C$,
- to stay in the queue in front of $L2$ since it is shared among other processing nodes,

- to be serviced by $L2$, i.e. an answer ans is generated and sent toward P,
- the answer ans to reach P travelling again the interconnection network $P - C$.

Instead, in the latter, P regains the control after a time that could be potentially much greater because aggravated by the impact of a second queue and because the request has to exit to the chip to reach M. At the end, we have to add the time for

- $L2$ to generate a request toward the main memory,
- the request to reach M travelling another interconnection network $C - M$ (not the same as before and eventually out of chip),
- to stay in the queue in front of M,
- to be serviced by M, i.e. a reply rep will be generated and sent back to $L2$

A way to model this behaviour is that clients still continue to model processing nodes, but the difference lies in more levels of server because we have to model more levels of shared memory. A classical view of the advanced client-server model taking into account the above assumptions is shown in Figure 7.1. We notice that every module $S_i,\ i = 1, \cdots, m$ is at the same time server towards its n clients and client towards its server S. It is important to keep in mind that we can generalize all this aspects. In particular, we could have other hierarchical levels, e.g. a third one, or we can specify a different number of clients for any hierarchical level. In other words, we can think about a classical client-server model in which clients are realized following a compositional approach, e.g. a client could be an entire client-server model.

At this point, we could follow two options:

1. to extend the analytical resolution summarized in Section 4.2 in such a way the hierarchy is considered
2. to use directly the PEPA formalism

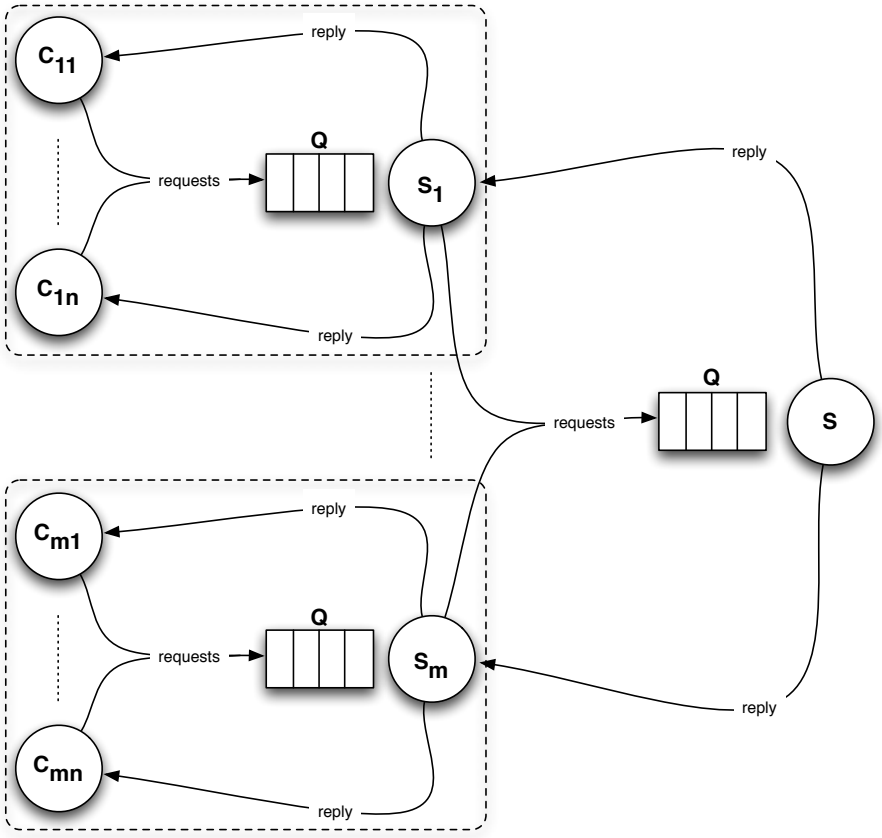


Figure 7.1: Hierarchical Client-Server Model with Request-Reply Behaviour

In principle, we were tempted for the first solution but some problems came out. The major obstacle is the increase in the complexity of the analytical resolution due to extensions needed to accomplish the goal so, at the end, iterative or numerical resolution techniques should be involved even then. Moreover, it is difficult to write the system of equations for a generic number of hierarchy levels. Currently, it is still an open problem to adapt the classical analytical resolution of the client-server model in order to satisfy server hierarchy.

Instead, exploiting the compositional property mentioned above for the model enhancement, a solution has been found in a easier way utilizing PEPA. In the next section, we are going to explain our contribution in this direction.

7.1.1 Definition

In the previous sections, we already told that it is possible to recognize statically the number of cache faults that will occur during the execution inspecting the sequential code. This brings to know also the frequency which the processing node generates memory requests, the so called $r_{request}$ (the inverse of T_P). It is worthwhile to note that in case shared memory hierarchies are involved, memory requests could be satisfied by any hierarchical levels. So a crucial point is to determine how many requests will be satisfied by a certain hierarchical level rather than another one. We are able to do this still by profiling.

We know that when a cache fault occurs, a memory request is sent toward the upper memory level. Suppose to have the architecture described above, the request is sent to the second level of cache $L2$. We can easily check statically if the requested block will belong to $L2$ or not. If so, we can consider $L2$ able to reply; otherwise the memory request will be forwarded to M. This way of reason allows to estimate the number of requests satisfied by any hierarchical level and, at the end, to find the probability to satisfy a request in a certain hierarchical level rather than another one.

For instance, suppose that the number of requests satisfied by $L2$ is c while m is the number of requests satisfied by M. Let p_c the probability to satisfy a request in $L2$ and p_m the probability to satisfy a request in M, we have:

$$p_c = \frac{c}{c+m}$$

$$p_m = \frac{m}{c+m}$$

In fact we have that, in average, p_c requests are satisfied by $L2$ while p_m are satisfied by M. At this point, we can use this information in the PEPA program to model a shared memory hierarchy.

The idea is to model processing nodes as clients able to generate requests toward either $L2$ or M. In other words, this means to have clients that can choose between two different actions, that are $request_c$ or $request_m$. Basically, this could be done in two different ways:

1. the first solution requires to write in a explicit way when a certain action must be taken by a client

2. the former capitalizes to the probabilities p_c and p_m to drive the choice between the two actions

Before to introduce the adopted solutions in depth, it is worth to note that differences between the two solutions lie exclusively in how clients are made. The other components of the system, i.e. the one modelling $L2$ and the other one modelling M, will be the same for both solutions. Briefly, we say that the PEPA component modelling the memory will be identical to the server already defined in previous chapters. Instead, the second level of cache plays two roles: from one side it acts as a server while on the other side it is a client. How we will see soon, this behaviour can be easily caught with PEPA.

First Version of Hierarchical Client-Server Model in PEPA We introduce the first solution through an example. For the time being, the parameter values are just constants. During the explanation of the test case, we will motivate choices about values. Suppose this scenario:

- 16 processing nodes, each one with rate $r_{request}$ of memory request generation. For simplicity, we are assuming homogeneous clients characterized by an unique phase. Of course, it is important to remark that the theory in Chapter 6 could be applied.

- as assumed previously, M is shared among all the processing nodes while $L2$ only among disjoint subsets of 4 processing nodes.

- $p_c = \frac{3}{4}$ and $p_m = \frac{1}{4}$.

As mentioned above, we want to write explicitly in the program when a processing node generates a request toward $L2$ (action $request_c$) or toward M (action $request_m$). The code below shows how this can be realized.

$$
\begin{aligned}
r_{request} &= 1.0/T_P \\
\\
P_1 &\stackrel{def}{=} (request_c, r_{request}).Pwait_1 \\
Pwait_1 &\stackrel{def}{=} (reply, \top).P_2 \\
P_2 &\stackrel{def}{=} (request_c, r_{request}).Pwait_2 \\
Pwait_2 &\stackrel{def}{=} (reply, \top).P3 \\
P_3 &\stackrel{def}{=} (request_c, r_{request}).Pwait_3 \\
Pwait_3 &\stackrel{def}{=} (reply, \top).P4 \\
P4 &\stackrel{def}{=} (request_m, r_{request}).Pwait_4 \\
Pwait_4 &\stackrel{def}{=} (answer, \top).P1 \\
\\
Cache &\stackrel{def}{=} (request_c, \top).(reply, r_{cache}).Cache + (request_m, \top).(ask, r_{ask}).Cache \\
\\
Memory &\stackrel{def}{=} (ask, \top).Memory + (answer, r_{memory}).Memory \\
\\
P_1[16.0] &\underset{request_c, request_m, reply}{\bowtie} Cache[4.0] \underset{ask, answer}{\bowtie} Memory[1.0]
\end{aligned}
$$

First of all, we notice that $r_{request}$ is evaluated in the same way as in previous sections. In fact, this is the general way to find it. A processing node is defined as a client composed by a sequence of states P_i followed by waiting ones, i.e. P_{wait_i}. In a generic state P_i a client performs an action that can be either $request_c$ or $request_m$. Instead, during a waiting state, a client can performs either a *reply* or an *answer*. In case a client, during a generic state P_i, performs the action $request_c$, it will wait for a *reply* in its following waiting state P_{wait_i}. Instead, if a $request_m$ is performed, the client will execute an *answer*. The length of this sequence, i.e. the parameter i, as well the frequency to perform a certain action rather than another one, is found looking at the probabilities p_c and p_m . In this case, $p_c = \frac{3}{4}$ and $p_m = \frac{1}{4}$ so we can define a client as a sequence of 4 states P_1, P_2, P_3, P_4 followed by the respective P_{wait_i}, $i = 1, \cdots, 4$. Three states will perform the action $request_c$ (followed by a *reply*) while the other one will perform a $request_m$ (followed by an *answer*).

The component *Cache* realizes the first server level, i.e. it is acting as $L2$. In case a request that it is able to satisfy is performed, it replies directly to the client executing a *reply*. Otherwise, it forwards the request to the upper

server level through the action ask. These actions have a own rate, that is the inverse of the service time required by the $Cache$ to perform them. It is worthwhile to say that this values are architecture details that can be easily found.

As already told, the component $Memory$ is the same of previous programs while the last expression defines the entire system.

Comments There are various considerations about the just introduced solution. First of all, it is important to say that the accuracy of the obtained results is very good as we will in the following section. In spite of this, some problems come out. The major constraint is just given by this explicit way to define the behaviour of a client. In fact, the sequence of interleaved $P_i - P_{wait_i}$ states is built taking into account the probabilities p_c and p_m. Up to now we were dealing with $\frac{3}{4}$ and $\frac{1}{4}$, so a sequence of length 4 can be easily written. Of course, the same approach would not be used if, for instance, $p_c = \frac{3}{17}$. The reason is simple and it does not belong in the complexity to write in PEPA the sequence but in how the underlying Markov chain is generated. In fact, longer sequences of actions bring to Markov chains with a more and more rigid structure. Since this property, the number of states composing rigid chains grows and to solve that chains becomes very expensive.

To accommodate any probability p_c and p_m and to keep reasonable the number of states forming the generated Markov chain, we need a more relaxed solution that we are going to explain in the next paragraph. Obviously, we will pay a price for this.

Second Version of Hierarchical Client-Server Model in PEPA The idea is to change the PEPA definition of client in order to drop the rigid structure coming out in the underlying Markov chain. Of course, all the other components will remain the same. This goal can be accomplished as shown in the following code.

$$
\begin{aligned}
P &\stackrel{def}{=} (request_c, r_{request_c}).P_{wait_c} + (request_m, r_{request_m}).P_{wait_m} \\
P_{wait_c} &\stackrel{def}{=} (reply, \top).P \\
P_{wait_m} &\stackrel{def}{=} (answer, \top).P \\
\\
Cache &\stackrel{def}{=} (request_c, \top).(reply, r_{cache}).Cache + (request_m, \top).(ask, r_{ask}).Cache \\
\\
Memory &\stackrel{def}{=} (ask, \top).Memory + (answer, r_{memory}).Memory \\
\\
P[16.0] &\underset{request_c, request_m, reply}{\bowtie} Cache[4.0] \underset{ask, answer}{\bowtie} Memory[1.0]
\end{aligned}
$$

A processing node is a component that can perform in a non deterministic way either the action $request_c$ or $request_m$. Of course, it is necessary to specify the rates of the actions, that are respectively $r_{request_c}$ and $r_{request_m}$. A way to do it is to evaluate the mean time between two consecutive requests toward $L2$ (tp_c) or toward M (tp_m) as

$$tp_c = T_P \cdot p_c$$

$$tp_m = T_P \cdot p_m$$

and finally to revert them for having the rates:

$$r_{request_c} = \frac{1}{tp_c}$$

$$r_{request_m} = \frac{1}{tp_m}$$

It is worthwhile to note that tp_c and tp_m could be estimated directly by profiling without to evaluate the probabilities p_c and p_m that will be instead used to evaluate T_{req} and T_{resp} as reported below.

7.1.2 Quantitative Comparison against the Simulation

Model Resolution The way to evaluate the under-load memory access latency in steady state condition of the system is basically the same as in the previous cases for both versions. Again, we base on Little's law to find out the so called $R_{Q_{server}}$. The difference lies in having more waiting states

and more incoming rates to that states. Anyway, we can easily to adjust the Formula 6.7 in this way:

$$R_{Q_{server}} = \sum_{i=1}^{w} \frac{p_{wait_i}}{\lambda_{reply} + \lambda_{answer}} = \frac{\sum_{i=1}^{w} p_{wait_i}}{\lambda_{reply} + \lambda_{answer}} \tag{7.1}$$

where

- p_{wait_i} is the average number of clients belonging to the state P_{wait_i} in steady-state condition of the system
- w is the number of waiting states. For instance it holds 4 in the example of the first version while it is equal to 2 in the last case.

Finally, as usual, we have to add the impact of interconnection structures for having the under-load memory access latency:

$$R_Q = R_{Q_{server}} + T_{req} + T_{resp}$$

It is important to recall that more interconnection structures are involved in hierarchical shared memory architectures, e.g. the already mentioned $P - C$ and $C - M$, so we have to take into account them in a proper way. The best solution is to consider again interconnection structures logically belonging to the server subsystem with the difference that the base network latencies T_{req} and T_{resp} are evaluated applying the definition of mean value:

$$T_{req} = T_{req_{P-C}} \cdot p_c + T_{req_{C-M}} \cdot p_m$$

$$T_{resp} = T_{resp_{P-C}} \cdot tp_c + T_{resp_{C-M}} \cdot p_m$$

Results We have simulated via JSIM and expressed through the two PEPA versions the following scenario:

- 16 processing nodes as in the previous tests
- M is shared among all the processing nodes and its service time T_S is exponentially distributed with mean value 29τ as in the previous tests

- $L2$ is shared among disjoint groups of 4 processing nodes. It is assumed to be big enough to contain all the working set needed to processes in execution over the same subset of processing nodes. Further, it spends in average 10τ to reply directly to the processing node with the requested cache block while it forwards the request to the upper layer in 4τ in average. Both these values are estimated taking into account second level caches with a proper size to be shared among various processing nodes.

- $p_c = \frac{3}{4}$ and $p_m = \frac{1}{4}$. These are constants chosen in such a way will be possible to evaluate the impact of the first hierarchical level of servers, i.e. the second level cache. Of course, it is important to note that, in real architectures, a second level of cache will not have a so high probability to fault because the use of techniques and optimizations at compile time, e.g. prefetching.

- T_P is our degree of freedom. It will take values in the range $[25\tau, 3000\tau]$. The reason to choose very low T_P values is quite simple: we are dealing with a primary cache so the average time between two faults is lower with respect to cases in previous chapters.

Figure 7.2 states the under-load memory access latency for the simulation (SIMULATION) and the two PEPA versions (respectively PEPAv1 and PEPAv2) while Figure 7.3 shows the absolute and relative errors of the two versions against the simulation.

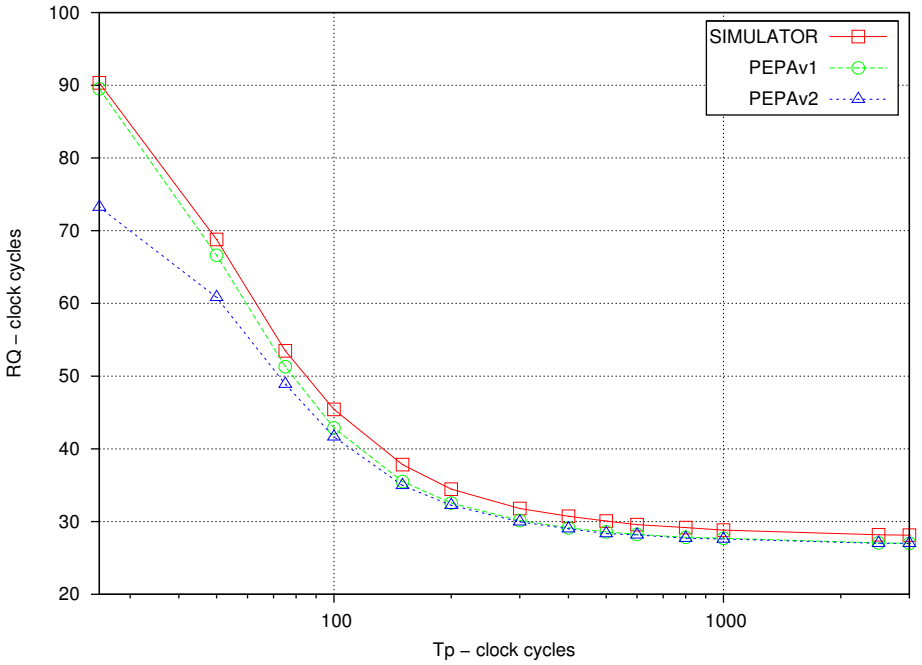
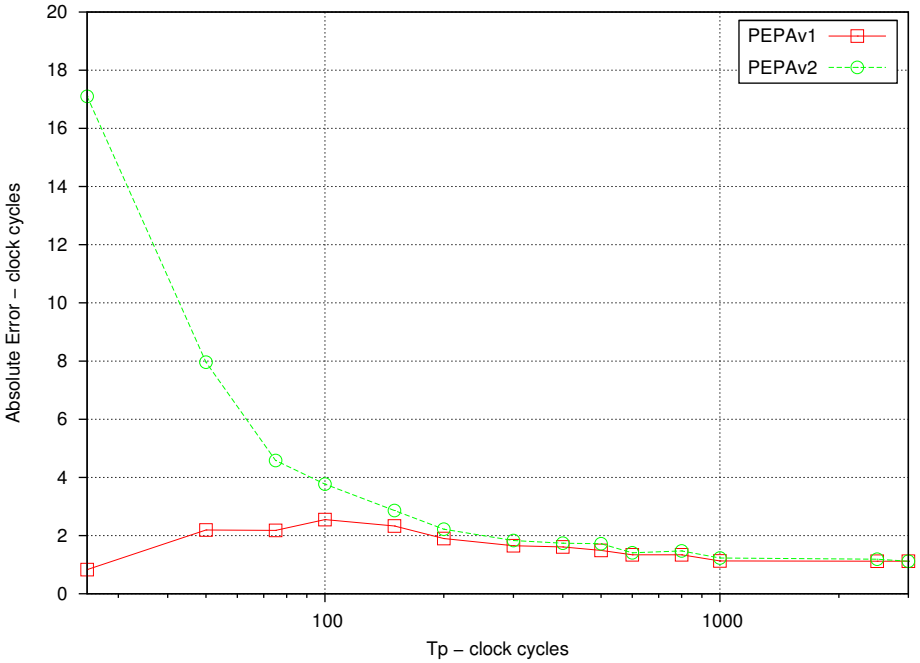
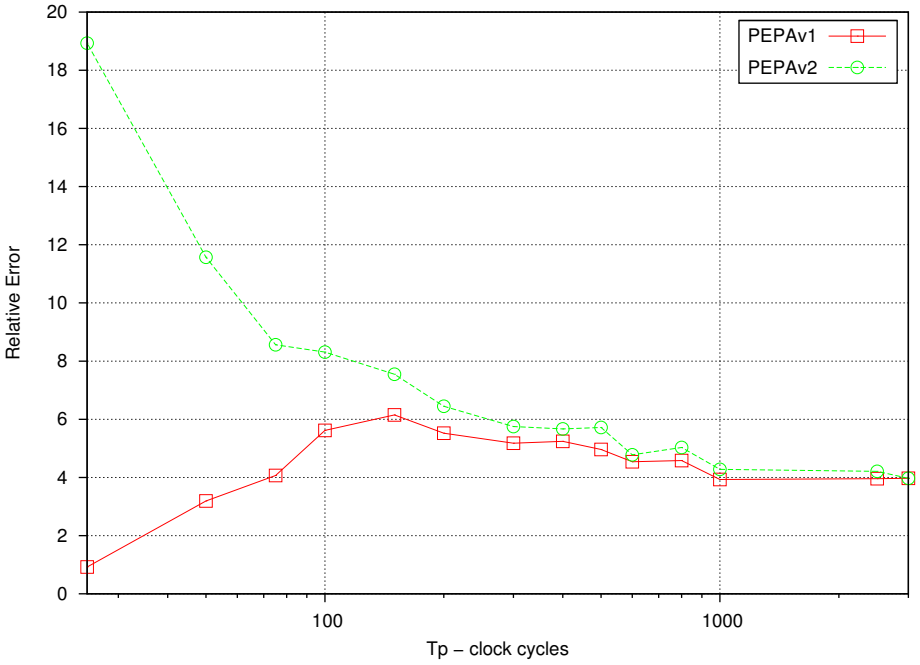


Figure 7.2: Under-load Memory Access Latency.



(a) Absolute Error



(b) Relative Error

Figure 7.3: Errors of PEPA with respect to the simulation.

First of all, we notice that both versions are underestimates of the simulation. The reason probably lies in the probabilist way to estimate some action rates. In confirmation to this, we have that the first version is very close to the simulation because it does not introduce much probabilistic behaviour. Instead, the second solution spaces out from the simulation only for lowest T_P. In fact, the shape of the second version approximates the simulation in a very good way for all T_P range unless the first two values, that are 25τ and 50τ. Therefore, for these T_P values, the maximum relative error is registered as reported in Figure 7.3(b).

On the other hand, we already know that the first version can not treat general cases because the structure of the generated Markov chains is very rigid and this is not suitable in terms of resolution. Instead, the second one is able to accommodate general cases without to enlarge the complexity. Therefore, we can conclude that the second solution is able to model hierarchical shared memory architectures in a good way being a trade-off between accuracy and complexity.

7.2 Conclusion

Due to the trend that multi-cores have been taken, to model hierarchical shared memory is becoming an important topic. In this chapter we have seen that this goal can be accomplished starting from the classical client-server model. Of course, enhancements are needed and they could be achieved in two principal way to operate: either extending the analytical resolution technique introduced in [20] or using the new formalism explained in Chapter 5 (PEPA). We decided for the latter because to extend the analytical resolution brings to a complexity increase that it is not worth. So we have seen two PEPA solutions on how is possible to treat hierarchical shared memory. On the base of tests that we have done, we concluded that the first one is very precise compared to the simulation but problems arise in terms of generated Markov chain. In fact, we have already told that the structure of the underlying Markov chain is rigid and this implies difficulty to solve it. On the other hand, the second version is a good trade-off between accuracy and complexity so it should be take into consideration.

Chapter 8

Conclusion and Future Works

Performance on shared memory architectures is dictated by the interrelation of concrete architecture details, parallel application constraints and run-time support of concurrency mechanisms. In order to exploit efficiently these systems is therefore necessary a methodological and structured approach to handle all this aspects.

On a hand, structured parallel programming is used in order to create parallel applications in an independent way from the underline architecture. Further, the use of a fixed set of paradigms to build parallel application allows optimizations, modularity and a methodology without increase the complexity as much. On the other hand, a cost model in association with an abstract architecture is needed from the performance point of view.

We have tried to give a contribution in this direction enhancing the cost model for shared memory architectures and its accuracy with particular care to parallel application constraints. The queuing based client-server model with request-reply behaviour has been our starting point. We have found other ways to express it with the goal to reach a formalism able to express flexibility, simplicity and a high-level approach. Of course, the resolution had to be a trade-off between complexity and accuracy.

We have decided for a Stochastic Process Algebra language (PEPA) as formalism to describe the Processors-Memory subsystem in an elegant way. Further, the numerical resolution technique is very accurate.

The next step has been to verify how PEPA was able to enhance the classical model. Therefore, in this thesis advanced cost models have been defined. In particular, we have focused on:

1. **shared memory hierarchies**. This architectural aspect is more and more frequent in multi-cores architectures so an advanced cost model was needed. The impact of more levels of shared memory could impact on the performance so, once a cost model is provided, a way to deal with this hierarchical organization could be found. Further, we believe that a parallel application organized in a hierarchical way could exploit these architectures in a very efficient way. So the hierarchical client-server model with request-reply behaviour could be a good starting point to study this topic.

2. **impact of the parallel application**. A first direct impact of parallel applications on the client-server model is to influence T_P values. We recall that T_P is an input parameter of the client-server model so its derivation is fundamental in order to have accuracy. In this thesis we have seen that this value is influenced in different ways: either due to complex internal behaviours of processes (the so called process phases) or for heterogeneity among processes.

   Following the structured parallel programming approach, heterogeneous processes are present only in some paradigms and usually their impact can be considered negligible with respect to other involved processes in the parallel application. However, a PEPA cost model for heterogeneous processes has been formalized in this thesis in order to be more precise from at least two point of view: to consider heterogeneous processes (and not to abstract from them) in addition to the accuracy of numerical resolution techniques. Further, we recall that this cost model could be used in an orthogonal way for dealing with process phases. It is worthwhile to say that the procedure of analysis in order to recognize different processes can be easily achieved looking at the structure of the application. Once processes are subdivided in classes is possible to introduce optimizations in order to reduce the complexity of the resolution technique.

   Also process phases are easily recognizable. Processes in structured parallel programming approach always interleave computational phases to inter-process communications and to establish when a *send* starts or ends is quite simple because these limits are software boundaries. The

difference in T_P between computational and communication phases can be even an order of magnitude while, within a computational phase, T_P can change but not so much. This makes sense to model only the so called *think* and *send* phases. Different approaches have been shown on how to deal with phases.

Further, it has been explained how phases-dependent cost models could be also used for processes not always working, i.e. their efficiency is less than one. It is worthwhile to recall that in these processes the T_P derivation can not be made only looking at the sequential code.

Bibliography

[1] http://en.wikipedia.org/wiki/dynamic_random-access_memory.

[2] Multicore development environment user guide. Technical report, Tilera Corporation, 2010.

[3] Tile processor i/o device guide. Technical report, Tilera Corporation, 2010.

[4] Java modelling tools, 2011.

[5] M. A. Arsan and F. Neri. *Elementi di teoria delle code, lecture notes in Teletraffic Engineering.*

[6] A. Clark, S. Gilmore, J. Hillstone, and M. Tribastone. Stochastic process algebras.

[7] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: a Hardware-Software Approach.* Morgan Kaufmann, 1 edition, August 1998.

[8] Richard Hayden and Jeremy T. Bradley. A fluid analysis framework for a Markovian process algebra. *Theoretical Computer Science*, 411(22–24):2260–2297, April 2010. Submitted to TCS, September 2008. Accepted 5 Feb 2010.

[9] J. Hillstone. Process algebras for quantitative analysis.

[10] J. Hillstone. *A Compositonal Approach to Performance Modelling.* Cambridge University Press, 1996.

[11] G. Iazeolla. *Impianti Reti Sistemi Informatici.* Franco Angeli, April 2004.

[12] A. Argent Katwala, J. T. Bradley, N. Geisweiller, S. T. Gilmore, and N. Thomas. *Modelling Tools and Tecniques for the Performance Analysis of Wireless Protocols.*

[13] L. Kleinrock. *Queueing Systems*, volume 1: Theory. Wiley- Interscience, 1975.

[14] Silvia Lametti. Modello dei costi delle tecniche di cache coherence nelle architetture multiprocessor. Master's thesis, Dept. of Computer Science, University of Pisa, Italy, October 2010.

[15] F. Luporini. Cost models for shared memory architectures. Master's thesis, University of Pisa, 2011.

[16] A. Papoulis and S. U. Pillai. *Probability, Random Variables and Stochastic Processes.* Mc Grow Hill, 4 international edition, 2002.

[17] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: the Hardware-Software Interface.* Morgan Kaufmann, 4 edition, November 2008.

[18] M. Reiser and S. S. Lavenberg. Mean-value analysis of closed multichain queuing networks. *Journal of the Association for Computing Machinery*, 27(2):313–322, April 1980.

[19] M. Tribastone, A. Duguid, and S. Gilmore. *The PEPA Eclipse Plug-in. Performance Evaluation Review*, volume 36. 2009.

[20] M. Vanneschi. *Architettura degli Elaboratori, and Course Notes of High Performance Systems and Enabling Platforms, Part 2, Section 3 - Master Program in Computer Science and Networking.* Pisa University Press, 2011.