University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Executable Performance Model and Evaluation of High Performance Architectures with Percolation

Adeline Jacquet†, *Vincent Janot*†, *R. Govindarajan*, *Clement Leung*, *Guang Gao*, *Thomas Sterling*‡

CAPSL Technical Memo 43
November 21, 2002

†Institut National des Telecommunications, INT, Evry, France.
E-mail: adeline.jacquet@int-evry.fr , vincent.janot@int-evry.fr
‡CACR California Institute of Technology, Pasadena, California, USA
E-mail: tron@cacr.caltech.edu

Abstract

Percolation has recently been proposed as a key component of an advanced program execution model for future generation high-end machines featuring adaptive data/code transformation and movement for effective latency tolerance. Percolation is related to conventional prefetch technique, but is more aggressive and smarter. A program unit (e.g. a procedure instance) is not ready to be scheduled for execution until the data it needs is in the right place (close to the code in the memory hierarchy) and in the right form (e.g. proper layout, etc.).

Supporting percolation is a major effort in the architecture design and the compiler/runtime software support. An early evaluation of the performance effect of percolation is very important in the design space exploration of future generations of supercomputers. However, performance evaluation of percolation using a traditional approach for computer architecture (e.g. execution-driven or trace-driven simulation) is both time consuming and impractical. Further, in early-stage architecture design/performance evaluation which deals with incomplete design details, or a program execution model with only a (sketchy) conceptual design, or an architecture without an (optimizing) compiler, make simulation-based approaches unsuitable.

In this paper, we develop an *executable* analytical performance model of a high performance multithreaded architecture that supports percolation. A novel feature of our approach is that it models the interaction between the software (program) and hardware (architecture) components. We solve the analytical model using a queuing simulation tool enriched with synchronization. The proposed approach is effective and facilitates obtaining performance trends quickly. Our results indicate that percolation brings in significant performance gains (by a factor of 2.7 to 11) when memory latency ranges from local memory access time to remote memory access time (in a multiprocessor system). Further, our results reveal that percolation and multithreading can complement each other and can work together to tolerate memory latency, especially in a multiprocessor system.

Contents

List of Figures

List of Tables

1 Introduction

This paper proposes a novel *executable* analytical performance model for early-stage performance evaluation of high performance multithreaded architectures [9] that support the fine-grain multithreaded program execution model with *percolation* [10]. Percolation is a smart and aggressive way to move and reorganize data/code to a proper level/space in the memory hierarchy before a program unit (e.g. a procedure, etc.) using these data/code can be scheduled for execution. Percolation facilitates the hiding of memory latency. Although the concept of percolation is somewhat related to prefetching, it is very different from prefetching. Further, percolation is orthogonal to and complements multithreading.

While the traditional simulation-based (trace-driven or execution-driven) performance evaluation approach is quite useful in obtaining performance predictions fairly accurately, it typically evaluates an architecture using a set of statically chosen application program or traces, and is not suitable to the kind of early-stage performance evaluation – the focus of this paper. First, such a simulation-based approach limits the performance evaluation to a fixed operating range since it deals with the details of a specific design and is not capable of exploring a large design space due to the huge cost of such simulations. Second, an implicit assumption is that an Instruction Set Architecture (ISA) design is complete and suitable system software design is available to realize the simulation. But, this is impractical in early stage performance evaluations of high-end machines, since we may face an architecture with incomplete design details, or a program execution model with only a (sketchy) conceptual design, or an architecture without an (optimizing) compiler.

The executable analytical performance evaluation model proposed in this paper, on the other hand, can evaluate architectures over a wide operating range and is helpful in identifying when certain architectural features are advantages and where they could be potential bottlenecks. Analytical performance evaluation approach also has the advantage that they are relatively less time consuming than the simulation approach. Although the analytical approach is less accurate than the simulation approach, analytical performance evaluation do provide a reasonable performance prediction under a wide operating range.

Although this paper is focused on the evaluation of a specific architecture model (e.g. percolation), the methodology proposed should have a much wider impact. It helps to evaluate future generation high-end machines, with many alternative architectures, program execution paradigms, and technologies. Further, it can help evaluate a larger design space consisting of architectures where the design is only at a conceptual level (i.e., not concrete) for one or more of the subsystems. The proposed approach can model and provide early assessments and feedbacks to the architects (both hardware and software) as to where the performance potentials or bottlenecks lie.

In this paper, we propose the use of an analytical model based on queuing networks [14, 13] for performance predictions. In particular, we develop analytical models for four different architectures, namely a single threaded single processor system, a multithreaded single processor system, a single threaded multiprocessor system and a multithreaded multiprocessor architec-

ture, supporting percolation. A novel feature of this work is that it models the architecture (hardware) and the software (program execution) components and the interaction between them. To accomplish this, we model program execution through a program graph that models coarse-grain parallelism in the application. The program graph is *executed* on the architecture model. In our study, we model the effect of long-latency memory operations and their impact on the performance. We study how percolating data from lower levels of memory hierarchy, such as remote memory, to high speed buffers or caches can improve the performance.

We solve the analytical model using a queuing network tool enriched with synchronization [8]. The executable queuing network model executes the (abstract) program model on the architecture to obtain performance metrics. This approach, referred to as the executable approach, simulates the execution of the program model on the architecture model. A major contribution of this paper is this integration of simulation approach and analytical models for performance predictions. The proposed approach is very effective in obtaining performance predictions for large complex systems fairly quickly. Further, it can obtain the performance predictions for a wide operating range. Such an approach is especially useful in design space exploration of high-end architectures where it is essential to quickly assess important architecture features.

Other major contributions of the paper are its performance predictions:

- Our results indicate that percolation brings in significant performance gains (by a factor of 2.7 to 11) in data-intensive and irregular applications, when memory latency ranges from local memory access time to remote memory access time (in a multiprocessor system).
- In large multiprocessor systems, where latency to remove memory is large, percolating even 50% of the required data can improve the execution time by upto 25%.
- Our results reveal that percolation and multithreading complement each other and work together to tolerate memory latency.

The rest of this paper is organized as follows. The following presents the necessary background on percolation. In Section 3, we describe the dynamic program model used in our study. The architecture considered is also described in this section. In Section 4, we present the analytical models for four different architectures. Section 5 is dedicated to the experimental results obtained with the performance models. A brief discussion on related work is presented in Section 6. In Section 7, we present concluding remarks and directions for future work.

2 Percolation

The concept of percolation was introduced by Gao et al., in [10] in the context of fine-grain multithreaded architecture. Percolation combines multithreading with dynamic and smart prefetching of context under the integrated control by programmer, compiler, runtime system and other related architecture layers. It is an aggressive and intelligent prefetching-like technique and is

useful in multithreaded as well as single-threaded program execution. In a mulithreaded architecture, percolation enables the movement of the thread context, including the activation frame, along with program instructions and control states, to higher level in the memory hierarchy. Percolation has two major objectives: reducing memory access latency by a smart distribution and programmable prefetching of data and providing a balanced distribution of data and program instructions by partitioning the application program into self-contained fragments. The movement (of context) is transparent and starts even before the execution of the thread (or the function containing the thread) corresponding to the context. Percolation can be *inward*, i.e., data moves towards where the procedure is to be executed, or *outward*, i.e., procedures move towards where data is available. In our study, we have only considered *inward* percolation.

As percolation moves data accessed by a thread to a higher level of memory hierarchy, it is likely to reduce the long memory latency which would other be experienced. In a non-threaded execution model, during this long-latency, the processor idles, waiting for the data to arrive from lower levels in memory hierarchy. In a threaded execution, if the long-latency operation is split across two threads (as in a split-phase read [12, 7, 9]), then the processor switches context during the long latency and executes other useful work. The data to be percolated is identified either by the compiler or by the programmer. A separate coprocessor, called percolation coprocessor, enables the movement of the context to higher level in memory hierarchy. Furthermore, percolation may involve data gathering/scattering as well as data reorganization within the memory hierachy.

Although percolation bears similarity to prefetching, percolation is different from software prefetching [6]. In percolation both data and the context of the threads are percolated, whereas in data prefetching only the data is prefetched. Further, percolation starts even before the corresponding thread, or the functional containing the thread, starts execution. Percolation can be handled by a separate percolation coprocessor. Lastly, percolation is orthogonal to multithreading and they complement each other. In this paper, in fact, we study to what extent they complement each other.

3 Architecture and Program Model

In this section, first, we present the program model used to model the software component. Next, we describe an abstract architecture that supports percolation in Section 3.2. In Section 4, we develop the analytical model for different architectures that we study in this paper.

3.1 Program Model

We model a program as a set of procedure instances p, characterized by a name and a list of dependencies. We refer to this as a program graph which consists of nodes, representing dynamic instances of procedures, and directed arcs connecting the nodes indicating dependencies between procedures. The program graph is acyclic, i.e., a procedure instance can only depend

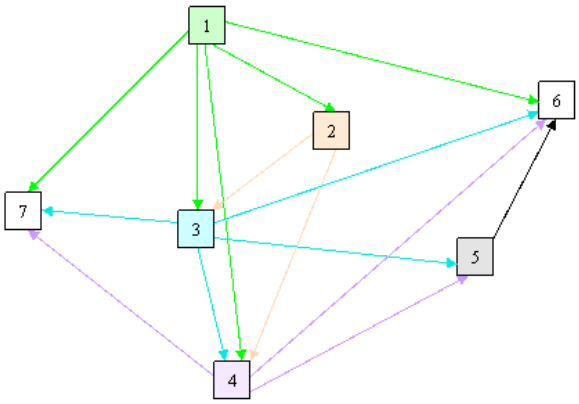

Figure 1: Example of dependencies between procedures in a program

on the previous instances of the same or other procedures. We assume that the number of dependencies, $nbparam(p)$, is an integer in the range 0 to 15 and follows certain distribution. In our experiments, we assume that the distribution followed by $nbparam(p)$ is as shown on Figure 2. The average value of $nbparam(p)$ for this distribution is 5.789.

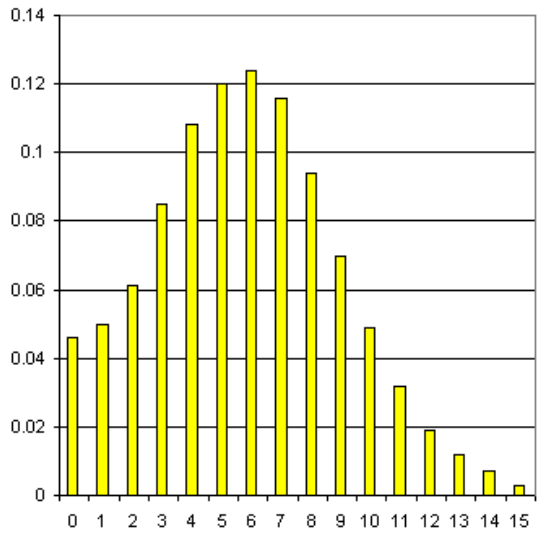

Figure 2: Distribution followed by $nbparam(p)$ and $nbresults(p)$

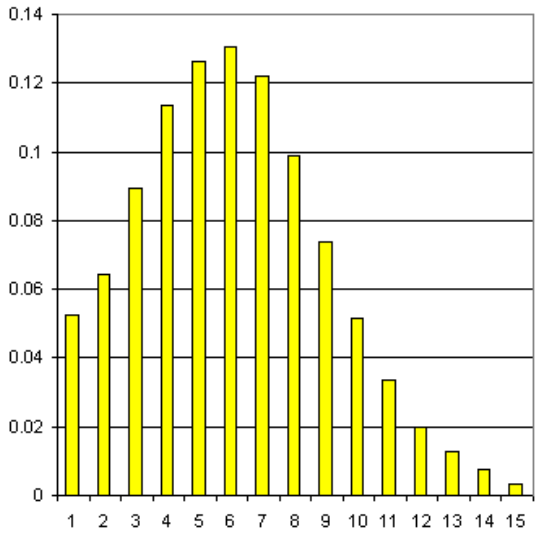

Figure 3: Distribution followed by $Size(p)$ for each instance of procedure

Each procedure instance is itself divided into several sequential parts. The number of such sequential parts is refereed to as $Size(p)$ in our model. $Size(p)$ is an integer greater than or equal to 1. In our experiments, we assume that the distribution followed by $Size(p)$ is as shown in Figure 3. The average value for $Size(p)$ is 6.069. Moreover, we assume that all these different parts are identical. The number of writes in lower levels of memory hierarchy is represented by $nbresults(p)$. We have assumed that the numbers of parameters and memory writes follow the

same distribution as the number of dependencies (Figure 2).

We generated a random program graph using the above parameters. To generate this graph, we assumed that each procedure is completed in one cycle and requires no data access to be executed. The parallelism profile of a random graph is depicted in Figure 4. The average coarse-grain parallelism in this program is 8.57. By varying the distributions for *nbparam*, *nbresults*, *Size*, and the number of procedures, we can obtain program graphs modeling a wide range of application characteristics. This program model is an input of the architecture model, which will be described in the Section 4.

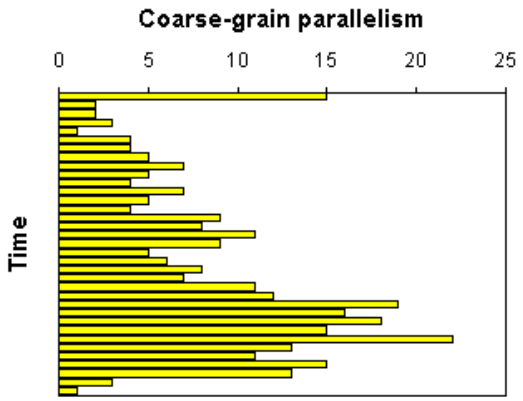

Figure 4: Coarse-grain parallelism of the program with 300 procedures

3.2 Architectures Considered in this Study

3.2.1 A Single-Threaded Single Processor Architecture

The architecture consists of four majors components as depicted on Figure 5, namely the synchronization processor, the percolation co-processor, the processor and the memory. We describe each of these components in detail below.

The Synchronization Processor (SP)

This component of the system is responsible for ensuring that all dependences of a procedure instance are satisfied. It takes the program model as an input. It enables the procedure instances, whose predecessors have finished their execution. Therefore, each time a procedure instance completes its execution (in the execution processor), the SP marks the dependencies, represented by outing arcs from this instance of procedure in the program graph, as enabled. An instance of procedure is said to be enabled, when all its input arcs are enabled. Thus, upon completion of each instance of procedure, the SP checks when any instance of procedure, not yet executed, can be enabled. An enabled instance of procedure is sent to the Percolation Co-Processor (PCP).

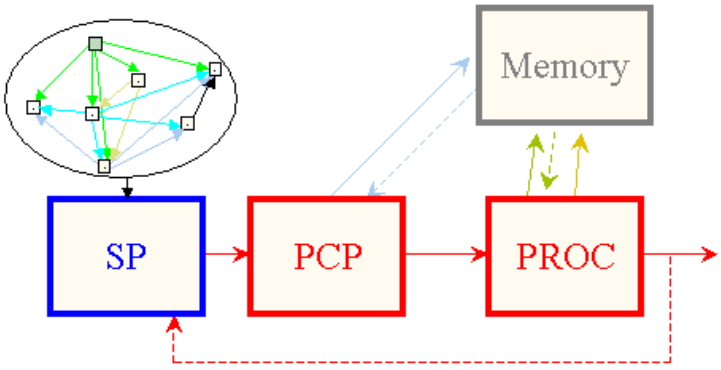

Figure 5: Architecture considered in this study

The Percolation Co-Processor (PCP)

The role of the PCP is to send requests to the memory to percolate data to higher levels in memory hierarchy. When all the percolation requests for a given instance of procedure are satisfied by the lower level memory (henceforth referred as Memory), the procedure is said to be *ready* for execution. A *ready* instance of procedure is sent to the processor for execution. Note that, in our model, we assume either a part or all of the data that is needed by an instance of procedure can be percolated.

The Processor ($PROC$)

The role of this component is to execute the *ready* procedures instances. The $PROC$ launches memory requests for those data that are not already available in the higher levels of memory hierarchy. As mentioned earlier, not all data can be percolated. Hence the $PROC$ launches requests for these data. When all the required data arrives in the high speed buffer or cache, the $PROC$ proceeds to execute the piece of code contained in the procedure instance. After this, it initiates $nbresults(p)$ writes to memory. However, the processor does not wait for the completion of the memory writes. Recall that each procedure is divided into $Size(p)$ parts. Hence the $PROC$ repeats the process of launching requests, execution of code, and result writes $Size(p)$ times. When all the sequential parts complete, we say that the procedure completes its execution and informs the SP.

The Memory

Memory is the lowest level in the memory hierarchy. We consider in this model a memory with $nbPorts$ ports. The Memory subsystem receives different types of requests: percolation requests, requests for data on cache misses (for data that is not percolated) and memory writes. Data requests concerning cache misses are treated with a high priority, although there is no preemption.

3.2.2 Multithreaded Single Processor Architecture

Multithreading has been proposed as a promising processor execution model to tolerate memory access latencies in parallel machines and as a mechanism to overlap the computation with inter-process communication [7, 9, 12, 17]. But most of the time, it cannot hide all the latency.

In this architecture, each sequential part of a procedure instance is treated as a pair $(Start, End)$ of threads. The first component of this pair, $Start$, initiates memory requests for data that is not in the cache (data that cannot be percolated). The $PROC$ then switches context to another thread. When the memory requests are satisfied, the second component of the thread, End, becomes ready and can be executed by the $PROC$. By switching context to another thread, the $PROC$ does not idle when the memory request is being satisfied. Combining multithreading and percolation is a good way to achieve higher performance and latency tolerance.

3.2.3 A Single-Threaded Multiprocessor Architecture

This architecture is the same as the non-threaded single processor described in Section 3.2.1, except the fact that we consider a multiprocessor and two levels of memory, a local memory and a remote memory. In our initial study, we have considered 4 processors in the system.

3.2.4 A Multithreaded Multiprocessor Architecture

This architecture is the combination of the previous ones, i.e., the single-threaded multiprocessor architecture described in Section 3.2.3 and the multithreaded single processor architecture described in Section 3.2.2.

4 Analytical Performance Model

In this section, we develop the analytical models for the different architectures. We develop a queuing network models for the different architectures in this section. In a queuing model, the main components are the queues, the servers and the customers visiting these servers for a service. Different types of distributions, e.g., exponential, deterministic, and hyper-exponential, are possible for the service time of a server. The exponential service time distribution is the most commonly used one, due to its memoryless property. The service disciplines followed by a server could be FIFO, LIFO, or priority-based. However, the discipline often followed by a server is FIFO, i.e. the first customer who arrives in the queue is served first. The number of customers at each server represents the state of a queuing network. An analysis of the state space of the queuing network yields the performance measures like the response time of the server [14, 13].

Obtaining closed form solution for complex queuing networks is difficult. This is more so when the underlying queuing model needs to support synchronization (of different customers)

at a queue. In our model, we need to ensure that a procedure instance p is enabled when all its predecessors have completed their execution. Similarly, it is ready for execution when all the percolation requests launched by a procedure instance has been satisfied. These activities require synchronization. To model them appropriately, we use a queuing network tool which can support synchronization [8]. Since (analytically) solving a queuing network model with synchronization is a harder problem [20], we resort to simulation of the queuing network to obtain the required performance metrics.

4.1 Modeling Data Percolation

We begin developing our analytical model, starting with percolation. A system contains certain amount of data located in several levels of memory. One of them is the cache which can be accessed in 1 or 2 processor cycles. Let *pcachebp* denote the percentage of data available in the cache before percolation and be useful for the computation of a procedure instance.

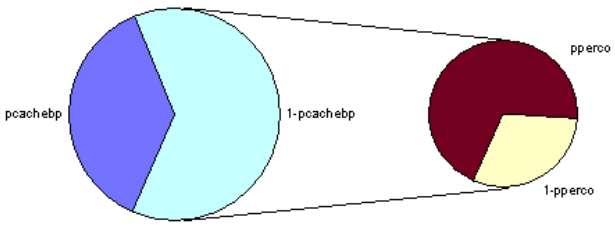

Figure 6: Modeling Data Percolation

Without percolation, *pcachebp* represents the percentage of cache hits. Therefore, $1 - pcachebp$ is the percentage of data not in the cache before percolation but required for computation. This required amount of data can be split in two parts as shown in Figure 6. The first part corresponds to the data that can be percolated and the second relates to data that cannot be percolated. Some data cannot be percolated because their addresses may be computed during the execution. We use *pperco* to refer to the percentage of data that can be percolated. Thus, the percentage of remaining useful data which is not in the cache and cannot be percolated is $(1 - pcachebp) \cdot (1 - pperco)$.

The list of symbols used in this paper is shown in Table 1.

4.2 Analytical Model for Single-Threaded Single Processor Architecture

In this section, we describe each part of the analytical model for a single-threaded single processor architecture.

The Synchronization Processor (SP)

We model the Synchronization Processor (SP) as a single server service center. At the beginning of program execution, there is one customer in the queue. When this customer

Table 1: Main parameters and acronyms

SP	synchronization processor
PCP	percolation co-processor
MEM	memory
MEML	local memory
MEMR	remote memory
PROC	processor
Size(p)	number of parts of a given procedure (number of threads in a multithread model)
nbparam(p)	number of parameters for a part of a given procedure
nbresults(p)	number of memory writes for a part of a given procedure
nbPorts	number of ports of the memory
nbProc	number of processors
pperco	percentage of percolated data
pcachebp	percentage of data in the cache before percolation
pload	percentage of load instructions
plocal	percentage of data in the local memory
premote	percentage of data in the remote memory
syncT	synchronization Time - Service time of SP
PercoPT	percolation preprocessing time - Service time of PCP
MemLat	latency of the memory
PercoLat	latency of percolation
C	context switching time
W(p)	execution time of a given part of a procedure as soon as all the data are in the cache
REQ	memory requests
REQP	memory requests for percolation
WB	memory writes

is serviced, it puts all the source nodes — nodes that do not have incoming arcs — of the program in input queue of the Percolation Co-Processor (PCP). Every time a procedure instance completes execution in the execution $PROC$, a customer is put in the input queue of the SP; when this customer is serviced by the SP, it checks whether any other procedure instances that are enabled and puts them in the queue of the PCP. The SP uses the program graph information to identify procedure instances that are enabled.

The Percolation Co-Processor (PCP)

When enabled procedure instances arrive from SP, they are serviced by the PCP one by one using a FIFO service policy. We assume that there are an infinite number of PCPs in our system. Servicing a procedure instance results in generating a number of percolation requests which are sent to the Memory subsystem. The number of percolation requests generated equals the number of data that are percolated. This depends on the number of input parameters to the procedure instance ($nbparam(p)$), the data that is not already in the cache ($1 - pcachebp$), and the extent of data that can be percolated ($pperco$). Thus the number of percolation requests sent to memory equals to $nbparam(p) \cdot (1 - pcachebp) \cdot pperco$. The percolation requests to Memory are of class $REQP$. The procedure instances wait in the PCP server until all the

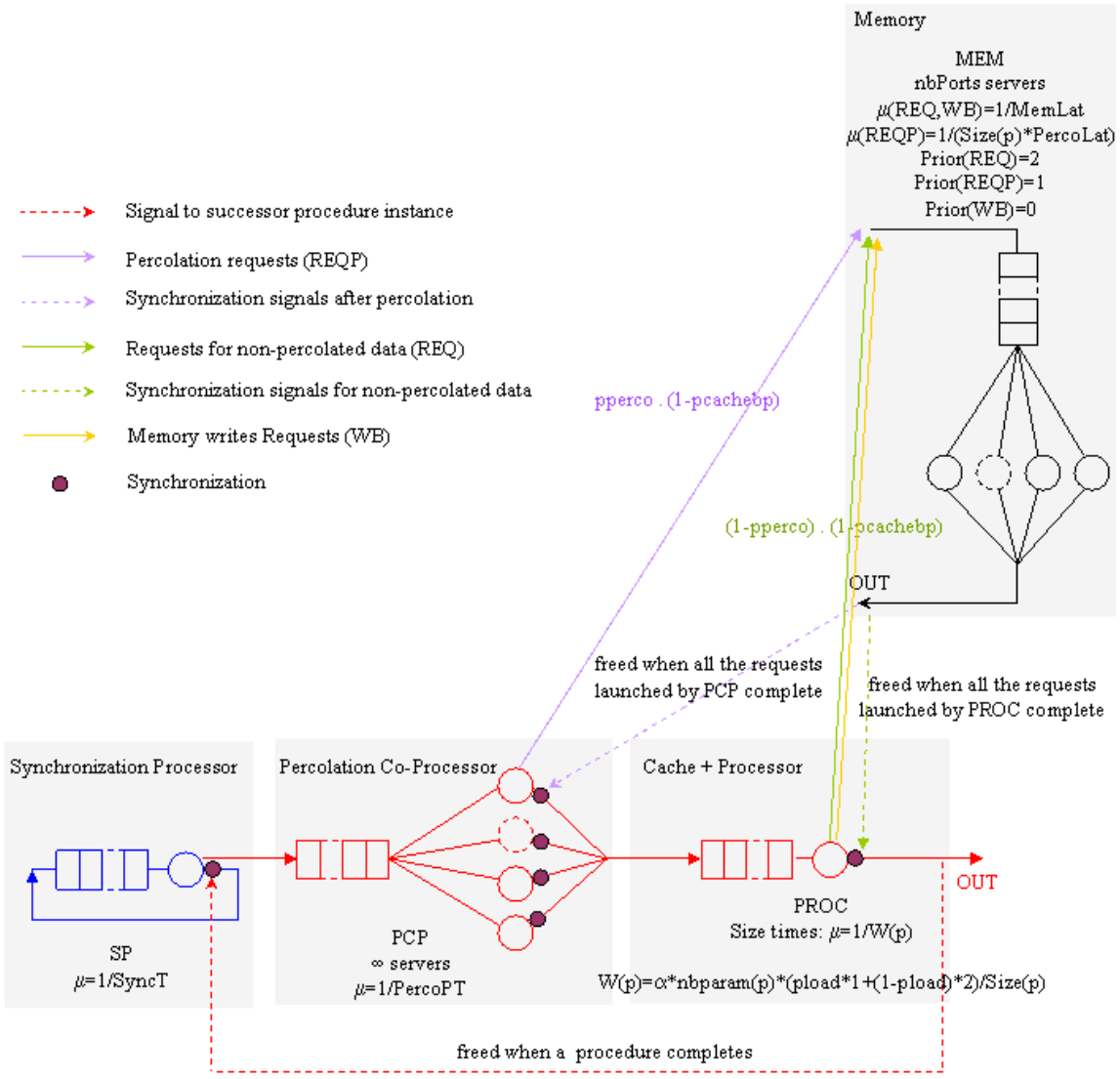

Figure 7: Analytical model for a single-threaded single processor architecture

percolation requests of the procedure instance are satisfied by Memory. This synchronization is represented in Figure 7 using a dotted arrow from the output of MEM to PCP. When all the percolation requests are satisfied by Memory, the procedure instance is sent to the processor ($PROC$) for execution.

We assume that the percolation preprocessing time for each procedure instance is an exponential distribution with mean $PercoPT$.

The Processor ($PROC$)

This component executes the ready procedure instances one by one. As mentioned earlier, in our program model, a procedure is a sequence of $Size(p)$ identical parts where each part consists of a set of data accesses (load instructions) followed by a set of other (non-load) instructions. For

the set of load instructions, the corresponding data may already be in the cache (the *pcachebp* part) or has been brought to the cache by percolation (equals $nbparam(p) \cdot (1 - pcachebp) \cdot pperco$). However for the remaining data, $nbparam(p) \cdot (1 - pcachebp) \cdot (1 - percop)$, which represents memory requests for the cache misses after percolation for a given sequential part of the procedure instance, memory requests are issued by the $PROC$. These requests are of type REQ and are sent to the Memory subsystem for service. Further, all the requests are sent at the same time without waiting that the previous data request is completed. The processor then waits until all these requests are satisfied by the Memory. This synchronization step is depicted in the Figure 7 by a dotted arrow between the output of the memory and the processor.

When all the data requests are satisfied by the MEM subsystem, the processor executes the set of non-load instructions in the sequential part. We assume that the number of instructions in a procedure instance follows exponential distribution with a mean $\alpha \cdot nbparam(p)$, for some constant α (In our experiment, we assume $\alpha = 100$). The probability of a load instruction is *pload*, and the cost of executing a load instruction is 1 cycle if there is cache hit. The execution time for other instructions is 2 cycles. Therefore, the execution time for a procedure instance is:

$$W(p) = \alpha \cdot nbparam(p) \cdot (1 \cdot pload + 2 \cdot (1 - pload))$$

Since a procedure instance p contains $Size(p)$ sequential parts, we assume that executing a part of an instance of procedure takes

$$W(p) = \frac{\alpha \cdot nbparam(p) \cdot (1 \cdot pload + 2 \cdot (1 - pload))}{Size(p)}$$

After the execution of this part of the procedure, $nbresults(p)$ requests for the memory writes are sent to the memory. These requests to memory are of type WB. Once again, all the write requests are sent to the memory at the same time. The $PROC$ does not wait for the completion of these memory writes.

When a sequential part of a procedure instance is completed, the processor $PROC$ executes, the remaining sequential parts in the same manner. The procedure instance completes when all the sequential parts have finished their execution. The $PROC$ then sends a completion signal to the SP which would help to identify other enabled procedure instances.

The Memory

We consider a memory subsystem with $nbPorts$ ports. Further, the memory system services requests using a priority order. However, there is no preemption. There are three classes of customers:

REQ: Requests of this class are launched by the processor when it requires to access a data item that has not been percolated to the higher levels of memory hierarchy. This class of requests has the highest priority in the memory system and has a mean service time of $MemLat$, the latency of the memory.

REQP: This class consists of percolation requests. It has a lower priority in the memory, compared to REQ requests. The service time for a $REQP$ request is exponentially

distributed and for a request belonging to a procedure instance p has a mean $PercoLat \cdot Size(p)$.

WB**:** Requests of this class are also launched by the $PROC$ unit and corresponds to result writes in the memory. This class has a lower priority than $REQP$ requests. The service time for a request belonging to this class has an exponential distribution with mean $MemLat$.

The complete analytical model for the simple single-threaded single processor architecture is shown in Figure 7.

4.3 Model for Multithreaded Single Processor Architecture

The architecture for multithreaded single processor is similar to the single threaded single processor architecture, the model for the former (refer to Figure 8) is also similar to that of the later. The main difference lies in the service of the sequential parts of a procedure instance. As mentioned earlier, each sequential part is considered as pair of *(Start, End)* threads. The *Start* thread initiates memory request for any data that has not been percolated. Instead of waiting in the $PROC$, the thread releases the processor resource and waits in a *Wait Queue*. When the memory requests are satisfied by the Memory system, the corresponding *End* thread is put in the *ReadyQueue*. The threads in the *Ready Queue* are executed one by one. Upon completion of the *End* thread, the processor launches result write requests to the memory. The service time for a sequential part of the thread remains the same as before and all of this time is attributed to the *End* thread. In addition, we assume that there is a context switch cost C, which is incurred before any thread starts execution. For the *Start* thread, we assume that the service time is equal to the context switch cost. By relinquishing the processor resource while waiting for the memory requests to be satisfied, the multithreaded architecture achieves better utilization of the processor and hence lower execution time.

4.4 Model for a Single-Threaded Multiprocessor Architecture

One of the main goals of percolation is to hide the memory latency. Since large systems with several levels of memory hierarchy and and/or multiprocessors have larger memory latency and more sensitive to such latency, knowing the impact of percolation on these systems is important. Hence we consider an architecture with two levels of memory, a local memory $MEML$ and a remote memory $MEMR$, as illustrated on Figure 9. The local and remote memory systems have the same number of ports $nbPorts$. Further the priority among the customers is also the same as that in a single processor system. The service time of the remote memory is 10 times that of the local memory. Thus, for a request launched by the processor (REQ or WB classes), the service time in the local memory is $MemLat$ and that in the remote memory is $10 \cdot MemLat$. For the percolation requests, the mean service times in the local memory and remote memory are $PercoLat$ and $10 \cdot PercoLat$ respectively. As before, these are mean values

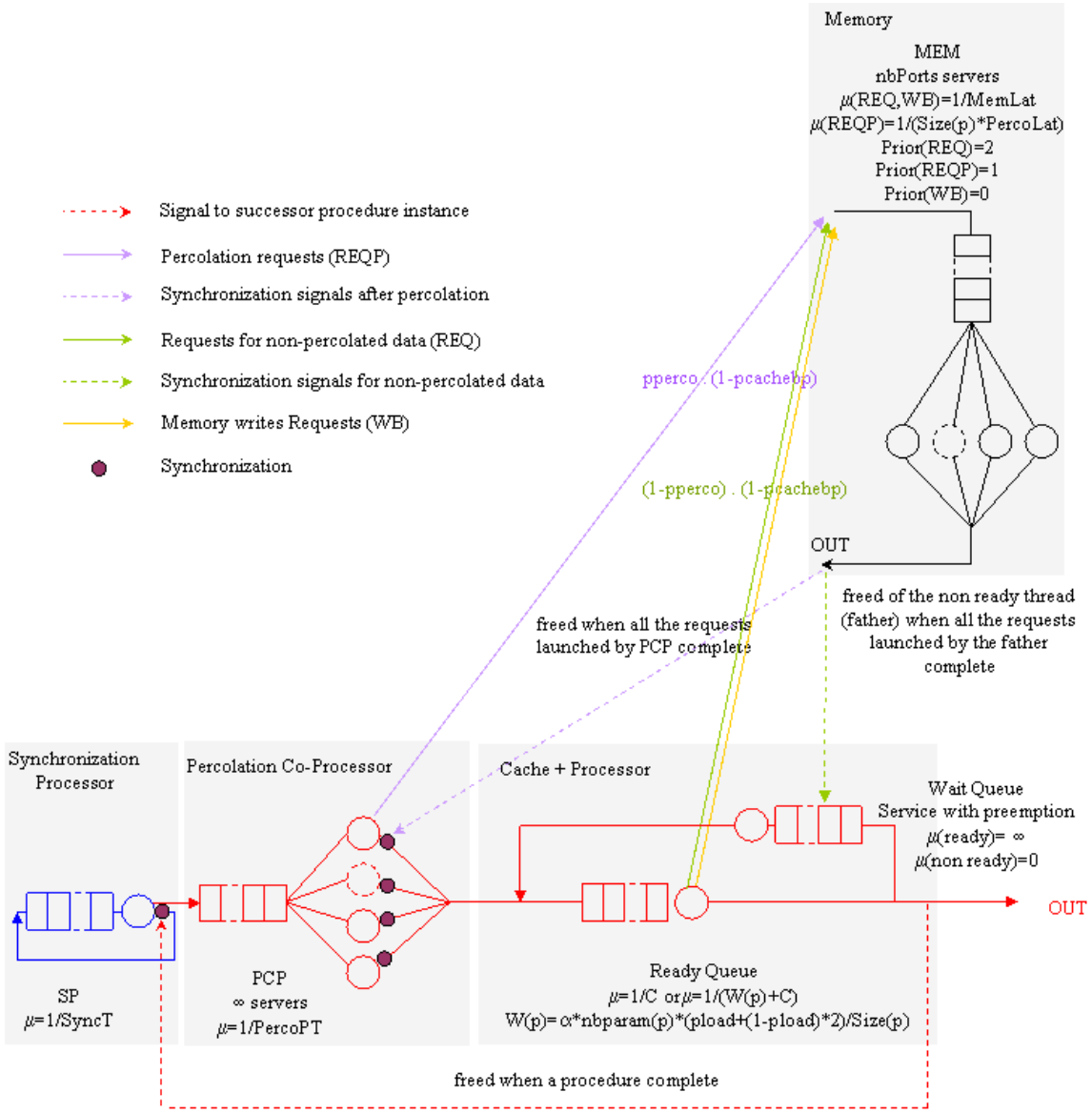

Figure 8: Analytical model for a multithreaded single processor architecture

and the service times are exponentially distributed. Further, we assume that *plocal* part of the requests are to local memory and $premote = 1 - plocal$ part to remote memory. Lastly, the processor has exactly the same behavior as in the single-threaded single processor model, but it has $nbProc$ servers instead of 1.

4.5 Model for Multithreaded Multiprocessor Architecture

This model is a combination of the single-threaded multiprocessor model and the multithreaded single processor model, and is illustrated in Figure 10.

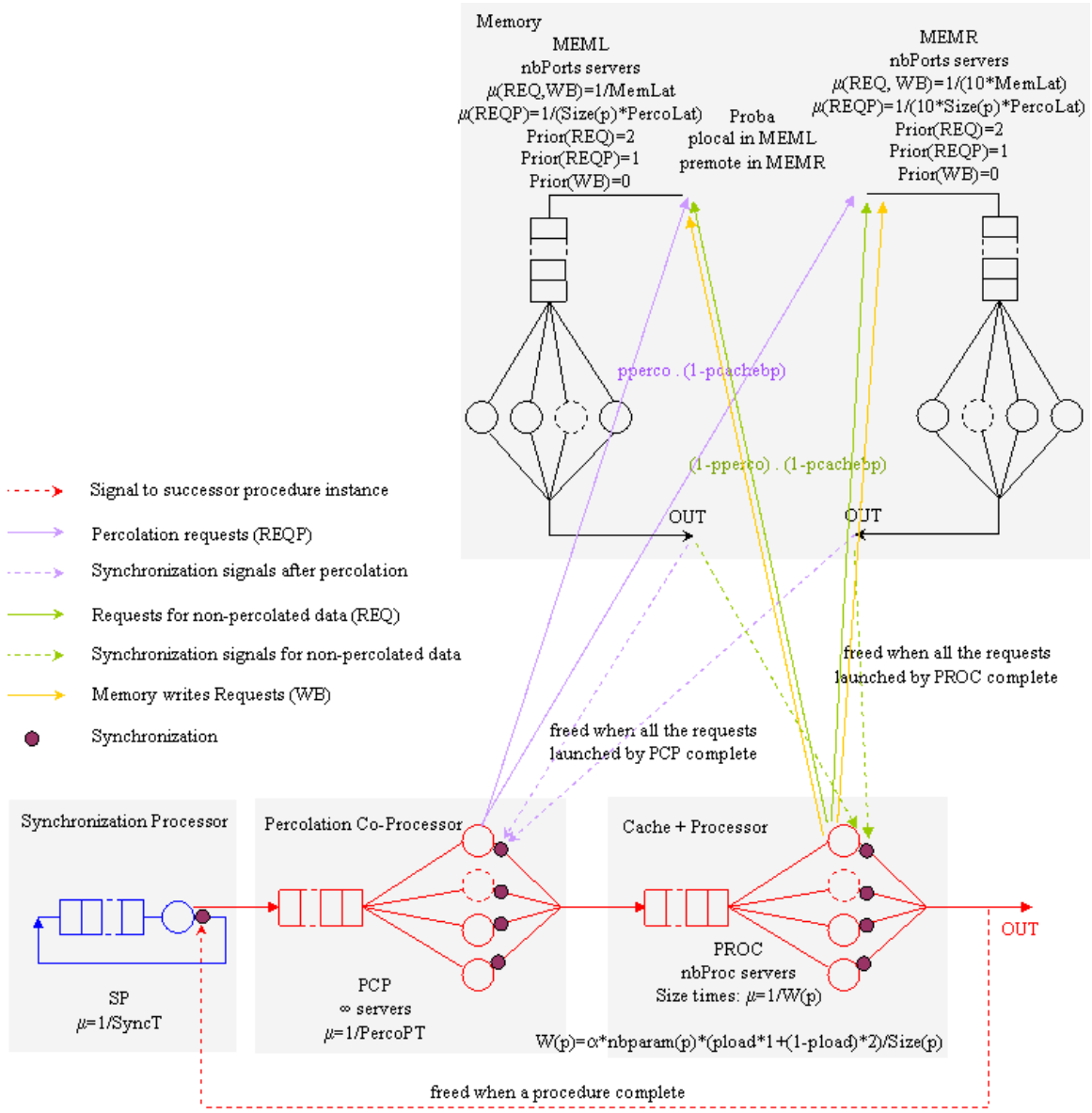

Figure 9: Analytical model for a single-threaded multiprocessor architecture

5 Experimental Results

In this section, first, we describe the experimental framework. Subsequently, in Section 5.2 we present a summary of our main results. Section 5.3 deals with detailed performance results for the various models.

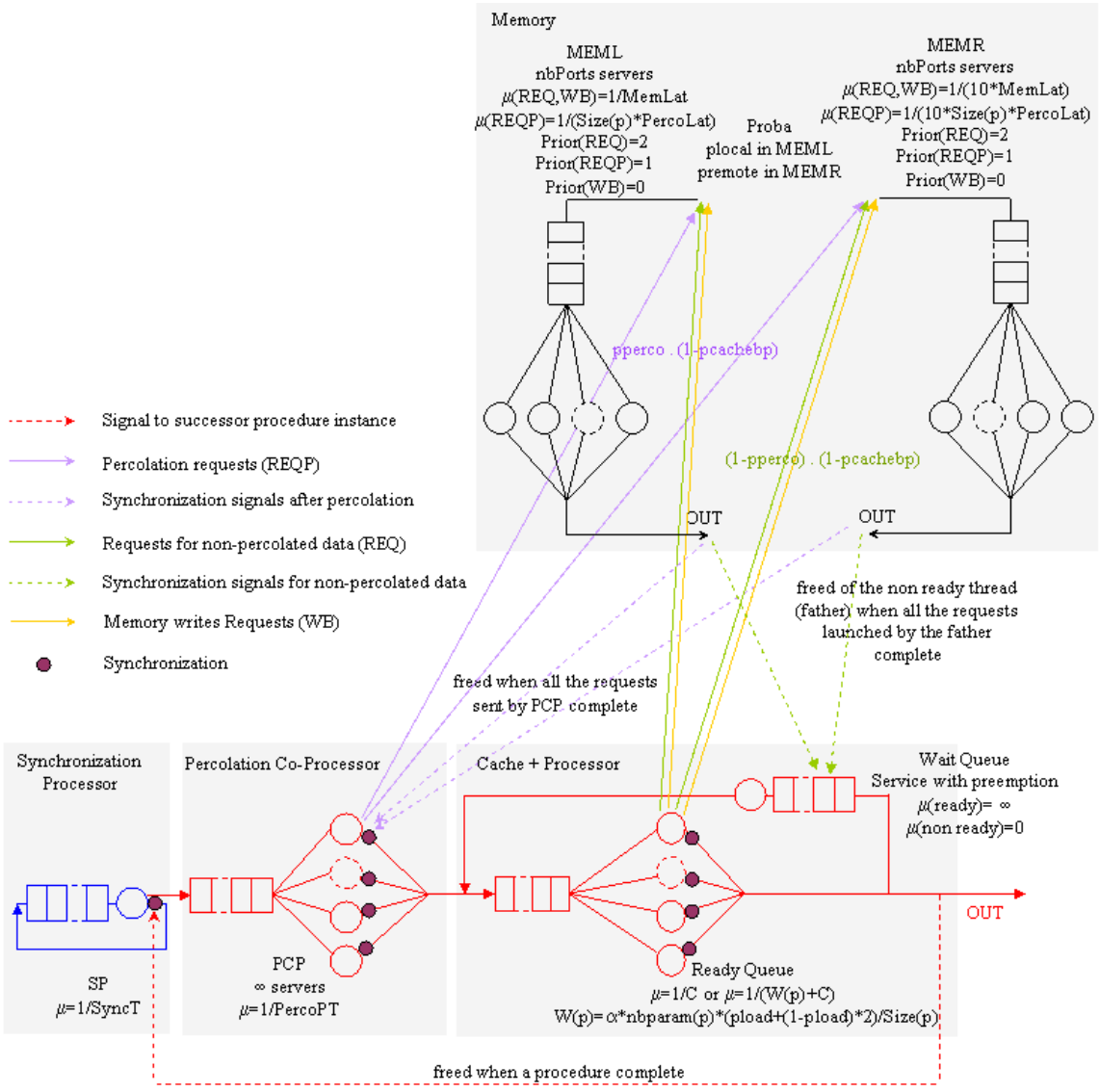

Figure 10: Analytical model for a multithreaded multiprocessor architecture

5.1 Experimental Framework

To solve our queuing network models we used a simulation tool called QNAP that simulates the queuing network [8]. This tool supports synchronization in queuing networks as it is required in our model.

The analytical model is specified by a number of input parameters. The value for some of these parameters, such as $MemLat$, $PercoPT$, and $nbPorts$, are kept fixed throughout our simulation. Table 2 summarizes those parameters whose values are kept constants in all our simulation. We vary a few parameters, such $pperco$, $pcachebp$, and $PercoLat$ to study the impact of percolation under a wide operating range. In our simulation, we use a single program

graph, consisting of 300 dynamic procedure instances, as the input program to the different architectures. To ensure that our performance parameters are obtained under steady state conditions, we ran a few experiments using a larger number of procedure instances (1500), and the variations were found to be very small.

The program graph structure, such as the dependences between the procedure instances, the available coarse-grain parallelism profile, and the number of sequential parts in a procedure, remain the same across different simulations in order to make fair comparison across different simulations run with different architecture parameters. The mean number of instructions per sequential parts and the mean number of instruction per procedure instance are constant across different simulations but their individual values are obtained from an exponential distribution. Likewise, the service time for the memory requests ($MemLat$ or $PercoLat$) also follows an exponential distribution and may not take the same service time for the same request in different simulation runs. To ensure that our simulation results are free of any stochastic fluctuations, we run our simulation for a specific set of input parameters a number of times (typically 50) and average the output performance parameters across these different runs.

Table 2: Values of the parameters

nbPorts	4
nbProc	4
α	100
pload	0.2
plocal	0.5
premote	0.5
syncT	100 cycles
PercoPT	50 cycles
MemLat	100 cycles or 1,000 cycles
PercoLat	$\frac{MemLat}{3}$
C	8 cycles

In our simulation, we have assumed a cache hit ratio before percolation (*pcachebp*) as fixed in each simulation, which corresponds to the hit ratio of a "warm" cache. In other words, the percentage of data available in the cache before percolation is fixed for each procedure instance in a single simulation. We supplemented our simulations with a set of experiments where the *pcachebp* value for each procedure instance is a varied within $\pm 20\%$, of the initial *pcachebp* value, but this did not show any significant variation in our results. Hence in all our experiments the cache hit ratio before percolation (*pcachebp*) is the same for all procedure instances in a simulation[1].

[1]However, the *pcachebp* value itself is a parameters and takes different values in different simulation runs.

5.2 Summary of Results

In this part, we will highlight our key points performance results.

- Our results show that percolation brings significant improvement in performance (by a factor of 2.7 to 11). The higher the memory latency, the greater is the gain in performance (refer to Section 5.3.1).
- Our experimental results indicate that percolation and multithreading can complement each other to hide latency. More importantly, percolation can improve the performance by more than 50%. These two techniques never compete. (refer to Section 5.3.2).
- Last, we notice that percolation is much more beneficial in the presence of long remote latencies as in a multiprocessor architecture (refer to Section 5.3.3).

5.3 Performance Results

In this section, we detail the performance results obtained from our executable analytical model.

5.3.1 Performance Improvement due to Percolation

First, we discuss the performance results for a single-threaded single processor system (refer to the analytical model shown in Figure 7). We evaluate the performance of the corresponding architecture for varying values of *pcachebp* and *pperco*. The execution times of our input program graph for different values of *pcachebp* and *pperco* and for low and high values of memory latency ($MemLat = 100$ and $MemLat = 1000$) are shown in Figures 11 and 12. As it can be seen from the graph, percolation brings increasingly significant performance gains when the cache hit ratios (before percolation) are low or the memory latency becomes large and unpredictable.

A lower cache hit ratio corresponds to irregular or data intensive applications. In these situations, the extent of improvement due to percolation is higher as the effects of (slower) memory are dominant. More specifically, when the memory latency is 100 cycles, and when $pcachebp = 0.1$, percolating all data can improve performance the execution time by a factor 2.7. When $MemLat = 1000$, complete percolation can reduce the execution time by an order of magnitude. However, we observe that percolating only 50% of the data decreases the execution time by a factor of 1.133 and 1.189, respectively, for $MemLat$ equals 100 or 1000! Lastly, we observe the performance trend is similar for a wide operating range of *pcachebp*, ranging from 0.0 to 0.8.

We observe similar performance improvement for single processor multithreaded architecture (refer to Figure 13) or under multiprocessor single-threaded architecture. The performance results for these architectures are shown in Figures 14 and 15.

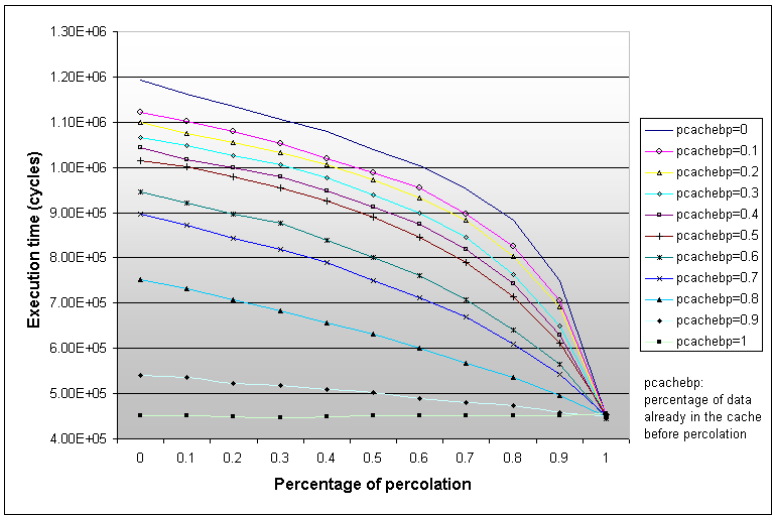

Figure 11: Execution Time with a low memory latency ($MemLat = 100$)

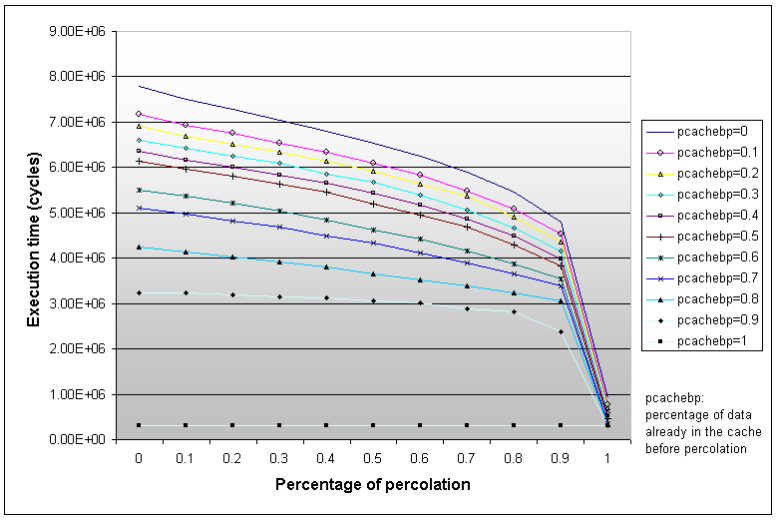

Figure 12: Execution Time with a high memory latency ($MemLat = 1000$)

5.3.2 Percolation vs. Multithreading

Percolation improves the performance even in a multithreaded system. With a coarse-grain multithreaded architecture which allowed us to hide a large part of the latency, percolation brings an additional improvement upto 20% (refer to Figure 13). As before, the lower the *pcachebp* value, the higher is the improvement. Further, we observe that multithreading complements percolation; even under no percolation (i.e., $pperco = 0.0$), multithreading reduces the execution time by a factor of 2 to 3. This is not surprising as multithreading is known to tolerate long memory latencies. But we observe that when percolation in conjunction with multithreading reduces the execution time even further.

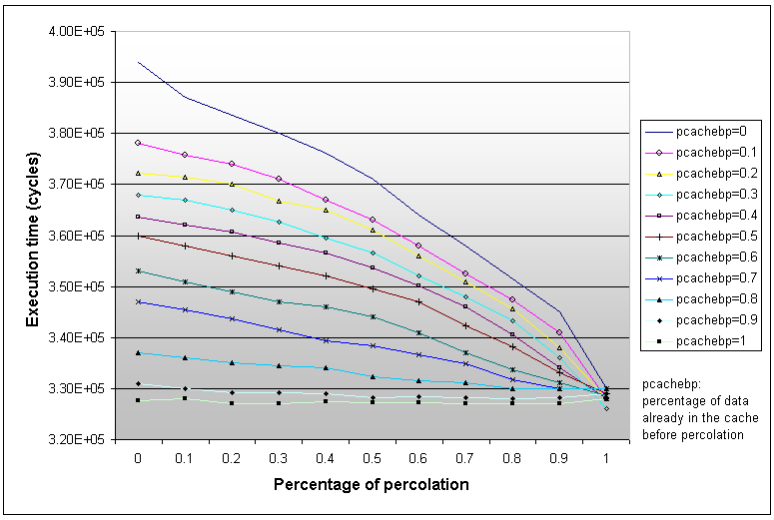

Figure 13: Execution Time with multithreading and $MemLat = 100$

5.3.3 Multiprocessor Systems

In the multiprocessor case, percolation increases the performance too. Percolation is much more efficient when we consider a multilevel memory, which is especially the case shown in Figure 9. Concerning the gains plotted on Figure 14, percolation divides the execution time by a factor 5.8 when $pcachebp = 0$ and $pperco = 1$ (full percolation).

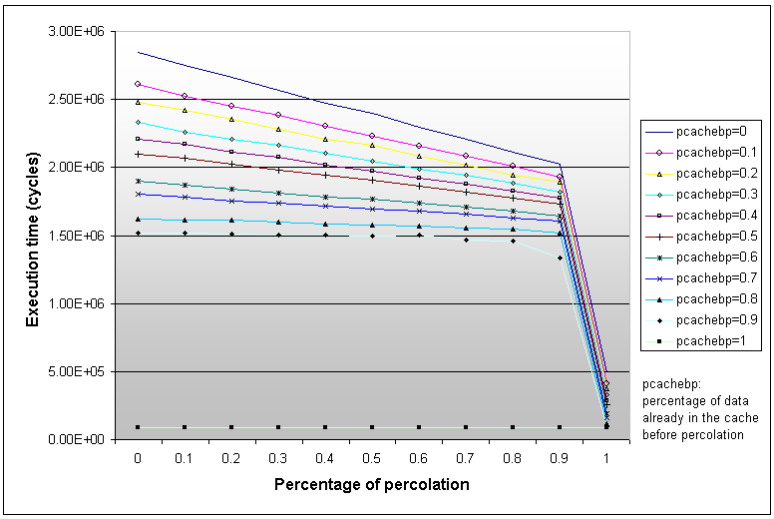

Figure 14: Execution Time with multiprocessors and $MemLat = 100$

Associating multithreading to a multiprocessor system allows to achieve a higher performance gain. Percolation brings an additionnal improvement by a factor of upto 3 (refer to Figure 15). Even for a moderate *pcachebp* value (say $pcachebp = 0.8$), the performance improvement due to percolation is nearly 50%. In this case, we observe that percolating only 50%

of the data decreases the execution time by 25% to 35% when *pcachebp* value is less than 0.8. To summarize, associating multithreading to a multiprocessor system allows us to achieve a higher performance gain. We observe that the complement each other to hide latency.

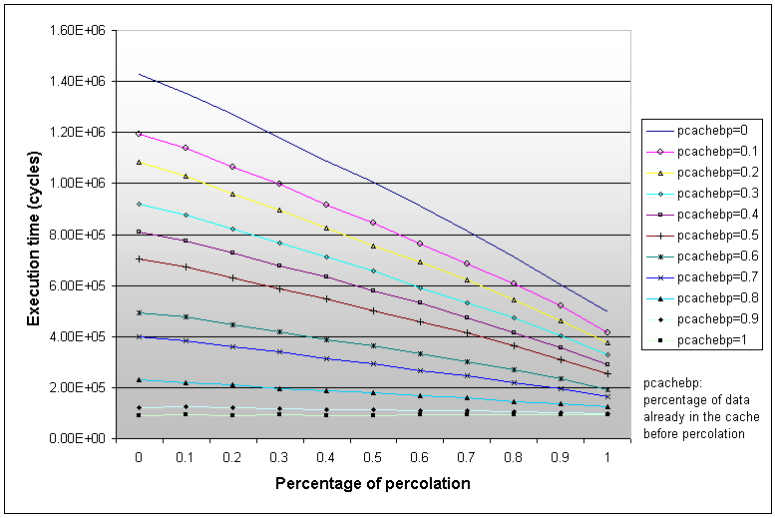

Figure 15: Execution Time with multithreaded multiprocessors and $MemLat = 100$

6 Related Work

A number of architectural proposals on the design of multithreaded architectures have been reported in the literature [2, 4, 7, 9, 11, 12, 17, 19]. The performance of multithreaded multiprocessor systems have been studied through an analytical approach by Agarwal [1], Alkalaj and Bopanna [3], Boothe and Ranade [5], Nemawarkar and Gao [16], Nemawarkar, et al. [15], and Saavedra-Barrera, et al. [18]. An essential difference between some of the earlier work and the work proposed in this paper is that it considers percolation and studies its effects on the performance. Also, it models the interaction of the software (program) component on the hardware more closely. Further, while the earlier analytical models based on queuing networks do not consider synchronization, our work models synchronization events more appropriately. Lastly, our approach aims more in predicting the performance trend rather than the actual performance.

7 Conclusions and Future Work

In this paper, we have developed an executable analytical performance model for high end architectures supporting complex program execution models such as multithreading and data percolation. We have evaluated the impact of percolation in different architectures. Our results indicate that percolation is an effective way to hide latency. It brings in significant reduction in program execution time (by a factor of 2.7 to 11), especially when the memory latency and/or

the cache miss ratio is large. Further percolation complements multithreading and brings in additional improvement (from 15% to 25%) in a multithreaded architecture.

Next, we discuss a few possible directions for future work. First, the current model is only for a single node multiprocessor system. Percolation is more beneficial in larger systems having hundreds of nodes, each having multiple processors. We plan to extend our model to study such a system. Anther major direction is to validate the proposed approach on the performance results, at least for the simple model, on a target architecture. To get more confidence in our performance prediction, we need to run our simulation for different sets of input program graphs. We also need to refine our models. In our current model, each procedure instance consists of a set of sequential components. These sequential parts act as a pair of threads in the multithreaded model. However, our program graph does not support any thread-level parallelism within a procedure instance. Further, in single threaded architectures, the parallelism available is only at procedure instance level (i.e., coarse grain). We need to investigate the impacts of percolation under fine-grain multithreading and on architectures supporting fine-grain parallelism. Also, we would like to model memory accesses in a more realistic manner. Currently all data accesses that are not percolated are launched at the same time in the *Start* thread in the case of a multithreaded processor or initially in the sequential part in the case of a single threaded application. In our model we have assumed our caches as "warm" caches. It would be interesting to study a more realistic cache model in which the cache are initially empty, and builds over a period of time during the execution. Lastly, the queuing network simulator used in our experiments has certain limitation in terms of the total number of customers it can support. This, to certain extent, has restricted us in running large simulations. We propose to either improve the tool or find/develop an alternative tool that could help running large simulations.

Acknowledgments

We acknowledge the finantial support of National Science Foundation, U.S.A. (NSF Grant no. 0103723). We are thankful to the Institut National des Télécommunications (INT), France, which gave the opportunity to the first and second authors to do their internship in the CAPSL laboratory. We would also like to thank Professor Monique Becker, INT France, for her technical support. The third author is on his sabatical from the Supercomputer Education and Research Centre Indian Institute of Science, Bangalore, India and acknowledges the support of his parent Institution. We wish to thank the members of the CAPSL team, University of Delaware, for their helpful suggestions.

References

[1] Anant Agarwal. Performance tradeoffs in multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, September 1992.

[2] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, Seattle, Washington, May 28–31, 1990.

[3] L. Alkalaj and R.V. Bopanna. Performance of multi-threaded execution in a shared-memory multiprocessor. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, December 1991.

[4] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Conference Proceedings, 1990 International Conference on Supercomputing*, pages 1–6, Amsterdam, June 11–15, 1990.

[5] Bob Boothe and Abhiram Ranade. Improved multithreading techniques for hiding communication latency in multiprocessors. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 214–223, Gold Coast, Australia, May 19–21, 1992.

[6] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Santa Clara, California, April 8–11, 1991.

[7] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–175, Santa Clara, California, April 8–11, 1991.

[8] Simulog France. *QNAP - Queuing Tool*. http://www.simulog.fr/.

[9] G. R. Gao., J-L. Gaudiot, , and L. Bic. *Advanced Topics in Dataflow and Multithreaded Computers*. IEEE Computer Society Press, 1995.

[10] Guang R. Gao, José Nelson Amaral, Andrés Márquez, Kevin B. Theobald, Sean Ryan, Zachary Ruiz, Thomas Geiger, and Christopher J. Morrone. HTMT phase 2 report. CAPSL Technical Memo 31, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, July 1999. In ftp://ftp.capsl.udel.edu/pub/doc/memos.

[11] Herbert H. J. Hum and Guang R. Gao. A novel high-speed memory organization for fine-grain multi-thread computing. In E. H. L. Aarts, J. van Leeuwen, and M. Rem, editors, *Proceedings of PARLE '91 – Parallel Architectures and Languages Europe,* volume I, number 505 in Lecture Notes in Computer Science, pages 34–51, Eindhoven, The Netherlands, June 10–13, 1991. Springer-Verlag.

[12] Robert A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 131–140, Honolulu, Hawaii, May 30–June 2, 1988.

[13] L. Kleinrock. *Queuing systems, Volume 1: Theory.* John Wiley & Sons, Inc., New York, 1975.

[14] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik. *Quantitative System Performance: Computer System Analysis using Queue ing Network Models.* Prentice-Hall, Inc., 1984.

[15] S. S. Nemawarkar, R. Govindarajan, G. R. Gao, and V. K. Agarwal. Analysis of multithreaded multiprocessors with distributed shared memory. In *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*, pages 114–121, Dallas, Texas, December 1993.

[16] Shashank S. Nemawarkar and Guang R. Gao. Measurement and modeling of EARTH-MANNA multithreaded architecture. In *Proceedings of the Fourth International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 109–114, San Jose, California, February 1–3, 1996.

[17] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: a killer micro for a brave world. CSG Memo 325, Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, Massachusetts, July 1991.

[18] Rafael Saavedra-Barrera, David E. Culler, and Thorsten von Eicken. Analysis of multithreaded architectures for parallel computing. In *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 169–178, Island of Crete, Greece, July 2–6, 1990.

[19] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, Santa Margherita Ligure, Italy, June 22–24, 1995.

[20] S. Varma. *Heavy and light Traffic Approximations for Queues with Synchronization Constraints.* PhD thesis, University of Maryland, College Park, MD, 1990.

A QNAP

A program in QNAP Code is divided into four parts

- Description of the system
- Declaration of the variables and objects
- Control information for solving a queuing model
- Execution of the model

It always ends with an $/END/$ intruction.

Each of the different parts begins with a key word. Therefore, a program in QNAP has the following structure:

$/DECLARE/$
: declaration of the objects types and the variables used in the program

$/STATION/$
: description of the topology of the queuing network, the different objects, the servers...
Each service station is described in this part by a $NAME$, a $SERVICE$, a $TYPE$ and a $TRANSIT$ parameter. $SERVICE$ can be a simple expression like EXP(time) or a piece of code to discribe more complicated behaviour. $TYPE$ usually defines the number of servers or specifies if the Station is a source station which just creates customers. $TRANSIT$ describes where to send the customers after beeing serviced in this service station.

$/STATION/$
: ...

$/CONTROL/$
: control of the program, choice between simulation and an analytical resolution, definition of the maximum runtime for one simulation...

$/EXEC/$
: initializing, launching of the program with the instruction $SIMUL$ and eventually data processing and display or storage of the results

$/END/$
: end of the program

When this program is executing, it begins by reading the $/DECLARE/$ part, then the $/EXEC/$ part. When the instruction $SIMUL$ is found in the code, according to what is in the $/CONTROL/$ part, the behaviour of the queuing system described in the $/STATION/$ part is simulated or solved analytically. As soon as the simulation is finished, the part following the instruction $SIMUL$ of the $/EXEC/$ part continues to execute until the $/END/$ instruction.