PARALLELISM-ENERGY PERFORMANCE ANALYSIS OF MULTICORE SYSTEMS

BOGDAN MARIUS TUDOR

B. Eng., UNIVERSITY "POLITEHNICA" OF BUCHAREST, 2007

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2013

DECLARATION

I hereby declare that the thesis is my original work and it has been written by me in its entirety.

I have duly acknowledged all the sources of information which have been used in the thesis.

No portion of the work referred to in this thesis has been submitted in support of an application for another degree of qualification at this or any other university or institution of learning.

Bogdan Marius Tudor
5 November 2013

Abstract

Modern multicore systems consist of multiple on-chip cores supported by off-chip shared resources such as memory and I/O devices. Scaling the performance in the multicore era requires programs to expose sufficient parallelism such that their execution consists of overlapping activities on both on-chip and off-chip resources. But too much overlap might trigger contention for the shared resources, extending the response time of the application. On the other hand, the performance of many multicore systems is increasingly constrained by either a power or an energy budget. Thus, in the multicore era, analyzing the performance of an application requires understanding of how the application parallelism is mapped to hardware parallelism, and how its exploitation affects the execution time and the energy usage.

This thesis proposes a hybrid measurement–analytical modeling approach for analyzing the performance of shared-memory applications on multicore systems. For a given application we predict the impact of the number of cores and core clock frequency on the parallelism and energy performance on traditional x64 and emerging low-power ARM multicore systems. The proposed parallelism model captures the overlap between response times of cores, memory and I/O devices to predict both the amount of parallelism exploited and the parallelism lost due to data dependency, memory contention and network I/O overhead. Based on the parallelism model and a static power characterization of a multicore system, our proposed energy model predicts the power and energy use of a program. In contrast to previous approaches that rely on instrumentation of the program source or binary code, our model uses non-intrusive inputs such as the size of the OS run-queue, hardware events counters and external power measurements. Validation against direct measurements of applications covering HPC, financial analysis, multimedia and datacenter computing on four UMA and NUMA multicore systems shows an average relative model error of around 6-13%.

A number of key insights are drawn using our approach. First, for memory- or I/O-bounded problems, allocating large number of cores increases energy usage and may also increase execution time due to resource contention among cores. Second, balancing the core and memory resources by selecting an appropriate number of cores and clock frequency can reduce the energy by up to 27% even on an ARM Cortex-A9 system. Third, we show that more energy savings can be achieved on datacenter workload *memcached* when balancing the cores, memory and I/O resources of a system by improving bottlenecked resources, rather than by turning off under-utilized resources. In summary, we show that balancing system resources is key to reducing the energy usage of an application, and this is achieved by improving the hardware performance, rather than by lowering the power usage.

Machines take me by surprise with great frequency.
– Alan Turing

Acknowledgements

This thesis was made possible by the supported received from many people throughout my (long) candidature.

I thank my PhD supervisor, Professor Teo Yong Meng, for the resolute support and for allowing me to carve my own research path. His passion as a researcher is only matched by his kindness and patience as a teacher. Through the six years of my candidature, I have learned from him numerous lessons that have permanently changed my outlook on research, teaching, professional life and far beyond. From him I learned the deeper meaning of *being a professor* and why this is perhaps the most philanthropic career one can embark on. Prof. Teo, I know I am responsible for many gray hairs from my clumsy writing, convoluted thinking and my many other issues, but I hope I can thank you by showing that your lessons have not been in vain.

I am indebted to many researchers that have helped me at different stages of the candidature. I thank my committee, Professors Tulika Mitra and Wong Weng-Fai from NUS and Professor Alan Edelman from MIT, for their comments and advices during the various milestones of my candidature. I thank Dr Verdi March for the numerous discussions on my research. Dr Simon See has helped me with many ideas and access to equipments. Professor Y.C. Tay has opened my eyes to the exciting world of modeling in his CS6282 lectures and has helped me greatly in the follow-up discussions when I inevitably got stuck. Professor Chin Wei Ngan has always been supportive of every equipment need I have had.

I thank my friends and colleagues in Singapore for making my time here a truly memorable experience. You are far too many to mention, but Cristi, Andrei, Marian, Claudia, Mihai, Narcisa, Marcel, Saeid, Andreea, Khanh, Dumi, Lavanya and Shi Lei have marked my time spent far away from home.

I thank my family for your efforts in raising me, for supporting my physical absence so I can pursue my dreams, and for pushing me to give out my best. I

feel their love and warmth every day, even when they are 9,000 km away.

Finally, I thank my wife Cristina for her love, unwavering support, encouragements and for inspiring me everyday to be a better person. Without her I would have not completed this chapter of my life and thus, it is fitting that to her I dedicate this thesis.

List of Publications

1. Bogdan Marius Tudor and Yong Meng Teo. *On Understanding the Energy Consumption of ARM-based Multicore Servers*, Proceedings of the 34^{th} ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pages 267–278, Pittsburgh, USA, June 17–21, 2013, acceptance rate 13%. **Paper is featured in HPCwire article *Mapping the Energy Envelope of Multicore ARM Chips*, 6 June 2013,** `http://goo.gl/Xvlth7`.

2. Bogdan Marius Tudor and Yong Meng Teo. *Towards Modelling Parallelism and Energy Performance of Multicore Systems*, Proceedings of the 26^{th} International Parallel & Distributed Processing Symposium PhD Forum, pages 2526–2529, Shanghai, China, May 21–25, 2012. **Awarded PhD Forum Best Poster Award.**

3. Bogdan Marius Tudor, Yong Meng Teo and Simon See. *Understanding Off-chip Memory Contention of Parallel Programs in Chip Multiprocessors*, Proceedings of the 40^{th} International Conference on Parallel Processing, pages 602-611, Taipei, Taiwan, 21–25 September, 2011, acceptance rate 22%.

4. Bogdan Marius Tudor and Yong Meng Teo. *A Practical Approach for Performance Analysis of Shared-Memory Programs*, Proceedings of the 25^{th} International Parallel & Distributed Processing Symposium, pages 652–663, Anchorage, USA, 16–20 May, 2011, acceptance rate 19%.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the end of the CPU frequency race, multicore systems have been pushed into the mainstream [10, 36, 48]. In the multicore era, the performance is scaled by increasing the number of cores that exploit the parallelism of a program. The execution on a modern multicore system consists of multiple activities involving both on-chip resources, such as cores and caches, and off-chip supporting resources, such as memory and I/O. An efficient execution overlaps the on-chip and off-chip activities, such that waiting time in the system is minimized. But with each technology generation, the gap between on-chip and off-chip performance is growing, leading to imbalances among the cores, memory and I/O resources. Understanding and mitigating this imbalance becomes critical because it impacts negatively the achievable execution performance and leads to large energy wastage in multicore systems.

In the multicore era, performance is scaled by exploiting parallelism. Thus, it is important to understand how much parallelism exists in a program and how much is exploited at runtime. A multicore processor consists of multiple parallel execution units called cores. The cores are pipelined and are often superscalar, and thus can execute multiple integer or floating point instructions, and issue

multiple memory and I/O requests at the same time. Thus, the execution of a program on a multicore system consists of overlapping multiple activities involving on-chip resources such as cores and caches, and off-chip supporting resources – such as memory and I/O. Through modeling the *parallelism performance* of a program, we can understand how many parallel units of work in a program can be executed on a number of cores, as a function of the off-chip supporting resources. Since the execution of a parallel program depends on many resources working simultaneously, its response time depends not only on the performance of individual resources, but also on the overlap between them. An efficient execution on multicore systems overlaps as much as possible the on-chip and off-chip activities, because that waiting time among different resources is minimized, which translates into a smaller response time of the program. But overlapping too many activities may hit the limits of off-chip resources and can introduce resource contention among cores. This may diminish the exploited parallelism and extend the execution time of the program.

In a multicore system, both on-chip and off-chip resources consume power. Since the energy usage of a program is proportional to its execution time and the power consumed by the resources, we can use the parallelism performance of a multicore as the handle for predicting the energy cost of a program. *Energy-proportionality* is a desirable property of a system in which the energy consumption is proportional to its useful work output. When a program incurs large waiting times among the execution resources, its energy proportionality will be low, because execution resources typically consume power even when idle. Understanding the parallelism performance enables predicting of execution configurations that balance the on-chip and off-chip resources, thus improving the energy-proportionality of multicore systems.

1.1 Motivation

With each new technological generation of multicore systems, the achievable parallelism performance and energy-proportionality are increasingly threatened by two problems: *utilization* and *dark silicon*.

The gap between on-chip and off-chip performance is increasing with each technological generation [74, 98]. Due to architectural constraints such as chip footprint, power and thermal issues, among others, cores need to share off-chip resources such as memory and I/O. This leads to competition for resources and contention among cores. For performance scalability, the number of cores is increasing with each technology generation but memory bandwidth is increasing at a much slower rate, because of wire delays and power dissipation, among others. Therefore, off-chip resources available per core are not keeping pace with the increase in the number of cores. Another trend is that memory capacity available per dollar continues to grow according to Moore's Law. Applications with larger program size executed on multicore systems result in bigger working sets, which in turn require larger off-chip bandwidth. As long as these technology trends continue, the overlap between on-chip and off-chip activities can be compromised, as on-chip resources are waiting for off-chip requests. Thus, scaling the number of cores introduces a *utilization problem*, when on-chip resources are less utilized compared to off-chip resources.

Increasing the number of cores introduces not only a utilization problem but also an energy-efficiency problem. Although the number of transistors that can be integrated into a chip continues to grows according to Moore's law, the scaling down of voltage supply, known as Dennard scaling, has stalled [36, 96, 97]. Due to this factor, the electrical power continues to increase across technological generations and thus, power density is becoming a serious performance issue because

multicore systems are entering the *dark silicon* era, where not all the transistors integrated onto a chip can simultaneously be used on a sustained basis due to electrical and thermal constraints [20, 38, 87]. Furthermore, this problem is exacerbated because many multicore systems are utilized not efficiently in terms of energy, especially when executing server workloads. In industry, availability and Quality-of-Service are even more important than utilization [16, 42, 43, 44]. The energy cost incurred by a program covers both cores and off-chip supporting resources. This introduces challenges and opportunities for better power management and for reducing energy wastage when the cores or the off-chip resources are poorly utilized. Considering that under-utilized servers still consume both IT and cooling power, this leads to significant energy wastage. It is estimated that the total power drawn by datacenters in the year 2012 amounts to more than 1% of total worldwide electricity consumption [35]. To make matters worse, more than half of this energy is wasted because of under-utilization, even in highly optimized data centers such as the ones run by Google [7, 81]. Thus, the race for better parallelism performance enters the energy-efficiency stage, where is imperative to understand the relationship between effective performance and its energy cost.

Coupled with the hardware shift to multicore, software is undergoing an evolution. Traditional parallel programming techniques such as Fortran, C or C++ supported by POSIX threads and OpenMP are joined by new software systems such as Cilk, Fortress, X10 [85], mobile applications frameworks, among many others. But new programming languages or programming models often have a high level of abstraction of the hardware, trading-off performance for programmer productivity. Furthermore, software has increasingly different resource requirements, including CPU, memory, I/O and graphics. But because of the programming abstractions, it is hard for the programmer to understand how effective is the overlap among the arithmetic operations and waiting for memory accesses and

I/O requests. Without knowing the degree of the overlap, software developers cannot understand the extent of the energy wastage in the system. Thus, there is a need to understand how the software parallelism matches the hardware parallelism of modern multicore systems, and how its exploitation affects the power and energy utilized.

1.2 Challenges

The changes in hardware and software systems pose new challenges in understanding the performance of multicore systems. With the growth in adoption of multicore systems, performance analysis of program parallelism increases in importance for several reasons. Firstly, at the design stage, the parallel program developer needs a method to understand the parallelism of the applications and of the execution performance. Modern multicore platforms are increasingly different in terms of core, memory and I/O performance. As such, the developers need to understand how their applications behave on different configurations, especially if they need to satisfy performance requirements such as a maximum execution time. Secondly, for users of multicore systems, performance evaluation allows them to understand which machine configuration satisfies their requirements. Thirdly, non-intrusive performance analysis methods can be applied concurrently to the program execution to auto-tune its performance. Finally, computer architects can optimize the design of a system if they understand how the parallelism of the program matches the parallelism of the system – potentially reducing both execution time and energy wastages.

However, the multicore shift brings a growing spectrum of parallel programming options such as programming models, languages, types of multicore systems, problem size, number of threads, number of cores, thread-to-core mapping and

memory architecture, among others. This leads to significant challenges in understanding the performance loss associated with each choice. With the wide adoption of multicore systems, there is a growing need for performance analysis methods that are general enough to be applied across both software and hardware platforms.

Current performance analysis approaches can be compared based on three key design trade-offs: ease of use, intrusiveness of the method and accuracy of the results. Recently, a shift in analysis methods recognizes that the performance of large parallel programs depends on a multi-dimensional space of options and configuration parameters. Therefore, the ease of applying the model across this parameter space is a crucial design criterion for performance analysis methods [28, 111]. Methods that rely on empirical data, such as regression based-approaches, neural networks and machine learning [28, 15, 41, 105] typically produce good accuracy but they require significant modeling effort or a large volume of training data. On the other hand, traditional methods for performance analysis include software instrumentation methods and trace-driven analysis [64, 65, 100]. However, while they have good accuracy, these approaches are intrusive and have a large cost of tracing. This reduces their applicability for big program sizes. Furthermore, instrumentation is often tailored to a particular programming language or binary platform. This leads to difficulty in generalizing them across programming languages and models or across different hardware platforms. Finally, analytical models are easy to apply, but often the simplifying assumptions about the hardware platform reduce their accuracy below practical usefulness. Furthermore, analytical model often do not use inputs which are easily available [8, 31, 34, 101]. With the adoption of multicore technologies in a wide range of systems, there is a need for performance analysis models that can be be applied across different programming languages and multicore architectures. Moreover, the need to scale

to large problem sizes can be met by non-intrusive analysis methods.

1.3 Objective and Approach

The key objective of this thesis is to develop a methodology for understanding and predicting the parallelism and energy performance of a given shared-memory application across different multicore architectures. Our methodology predicts the achieved parallelism and energy performance and performance loss caused by data dependency, memory contention among cores and I/O overhead.

To achieve this objective, we propose an analytical approach supported by observations derived from measurement experiments. Using measurement analysis of program execution on large multicore systems, we discover key insights on the causes of parallelism loss. Based on these insights, we propose an analytical model for the speedup and energy requirements of shared-memory programs, and the speedup loss due to data-dependency, memory contention among cores and I/O overhead. Using our model, users and developers of parallel programs can analyze and optimize parallel program executions, by predicting configurations that maximize the speedup, minimize energy requirements or achieve a trade-off between speedup and energy use. Furthermore, our model can be applied to determine the utilization of key resources, establish the system bottleneck and act as a guide in improving the energy proportionality of the system by balancing the core, memory and I/O resources.

Our analytical model of parallelism and energy performance relates the inherent and exploited parallelism of a program to the useful work and overheads of the program. The analytical model is divided into two main components: (i) a parallelism performance model and (ii) a power and energy performance model.

The parallelism performance model predicts the achieved speedup, the inherent

parallelism of a program, and the speedup loss. We address the speedup loss due to data-dependency among threads, memory contention among cores on UMA and NUMA systems and network I/O contention. To generalize this model across different software and hardware platforms, we use two non-intrusive and widely-available sets of metrics reported by modern systems: the dynamic size of the operating system run-queue as the proxy for program parallelism, hardware events counters as generalization of different hardware platforms to study the memory contention and a trace of network I/O operations for programs that have I/O requirements.

The power and energy performance model predicts the power and energy use by the execution of a shared-memory program. The power model uses a static characterization of the power drawn by the different system components and the hardware events counters to model the resource utilization.

The diverse workloads used include HPC dwarfs, real-world parallel applications and server workloads such as in-memory key-value store, which are widely used in datacenters by companies such as Facebook, Twitter or Amazon, among others. The target architectures are commodity Intel/AMD server systems and low-power ARM Cortex-A9 multicores. We target programs with large compute, memory requirements or network I/O requirements, and therefore we do not address workloads where the performance impact of storage I/O is crucial.

Next, we discuss the approach for applying the model. For an application and a target platform, we perform a small number of baseline executions. During these executions, which are conducted on a small number of cores, we collect two types of input parameters: (i) workload parameters and (ii) system parameters. The workload parameters are considered workload-dependent and must be collected for each application on a target system. The system parameters are considered system-dependent and are collected only once. Using these traces as the inputs of

our analytical models, we can predict the parallelism and energy performance for a given program on different number of cores and memory configurations.

1.4 Thesis Contributions

This thesis consists of two key contributions.

1. Performance approaches for parallelism in the multicore era.

   (a) Analytical models supported by hardware events counters — We develop an analytical model for understanding the parallelism performance of large shared-memory applications on traditional and low-power multicore systems. To improve modeling accuracy, we exploited traces of the operating system run-queue and hardware events counters as inputs for the model. To the best of our knowledge this is the first time the run-queue size is used as the proxy to program parallelism. The advantages are improved model accuracy, ease of use and generality with respect to the parallel programming languages and models supported [114].

   (b) Parallelism-Energy Performance — We propose an analytical model for the energy performance of a parallel application on multicore systems. As multicore system scale, the power or the energy budget becomes an important limiting factor on achieved performance. When the memory or the I/O becomes the system bottleneck, it leads to high energy cost and low performance. By relating the energy with the parallelism performance, our approach allows us to estimate the knee clock frequency that balances core and memory performance in multicore systems, as well as the I/O bandwidth required to sustain the core-memory performance. The novelty of our model stems from modeling the overlap among cores, memory and

I/O response times, and predicting the parameters that balance the system performance [115, 116].

2. Insights into the causes, and mitigation strategies for performance loss.

   (a) Parallelism loss due to memory contention — We provide insights on memory contention among cores, using experiments on state-of-the-art UMA and NUMA systems, with up to 48 cores. In contrast with previous studies [63], we show that memory burstiness depends on problem size. Small problems generate bursty accesses, but large problem sizes exhibit a non-bursty pattern of memory accesses when the program is slowed down by memory contention among cores [117, 116].

   (b) Energy-proportionality optimizations by adjusting existing configurations — We optimize of the execution of parallel applications by determining the number of cores and core frequency that achieves an optimal point in the power-performance space. The optimality criteria range from fastest execution time to minimum energy usage. The optimizations achieve significant energy reduction and performance improvements compared to the default OS scheduling policies [114, 116].

   (c) Energy-proportionality optimizations by improving the hardware — We show that increasing memory and I/O bandwidth can improve both the execution time and the energy usage of server workloads on low-power ARM Cortex-A9 systems. Counter to intuition, we show that restoring system balance by improving the bottleneck devices achieves larger energy savings compared to slowing down unutilized resources [116].

1.5 Thesis Organization

The thesis is organized as follows.

Chapter 2 presents related work in the area of parallelism performance and power-energy prediction. We discuss the classical approaches for parallelism performance and why their limitations in modern multicore systems require an extension of the classical definitions of inherent parallelism. In the area of power and energy prediction, we discuss the technical approaches for understanding the power used by a system, including simulations and analytical models. Lastly, we discuss the state of the art in energy-proportionality studies and their limitations.

In Chapter 3, we present the general models for parallelism and energy performance. The chapter is structured in two parts. First we discuss the general model for parallelism performance, and define the inherent and exploited parallelism of a program, as well as the parallelism loss. We show an implementation of the general model of parallelism performance for shared-memory programs with network I/O operations, and propose three sub-models for understanding the parallelism loss due to data-dependency among threads, memory contention among cores and I/O overhead. The second part the chapter discusses the proposed model for energy requirements, which is derived from a static characterization of the power requirements and the modeled service times of both on-chip and off-chip resources.

Chapter 4 discusses the observations from measurement analysis, followed by the parameterization of our model using baseline runs and by the validation of the model against measurements. First we discuss a series of measurements on state-of-the-art UMA and NUMA multicore systems with up to 48 cores and eight memory nodes. These observations simplify the modeling of the memory contention overhead. Second, we discuss model parameterization, the input parameters selections during the baseline runs and the static power characterization of the

system. Lastly, the chapter shows validation results of our model against direct measurements of exploited parallelism and energy usage on four traditional and one emerging low-power multicore systems.

Chapter 5 presents three applications of our model. The first application represents a typical problem in many datacenters today: determining the minimal configuration needed to execute a program within a pre-specified time. We show that by default an OS may chose configurations that lead to time and energy wastage, when the memory is the system bottleneck. Second, we apply the model to predict the number of cores and the core frequency that minimizes the execution time of a program execution, and show that important energy savings can be achieved by turning off unutilized resources, compared to the default OS allocation policies. Third, we present a method for system architects to improve the energy proportionality of low-power multicore systems using directed power allocation. We show that our model can be used to direct a power allocation strategy to improve the performance of bottleneck devices. We show that more energy is saved by increasing the performance of bottleneck devices than by turning off unutilized devices. This shows that the key for improving energy proportionality is higher performance, even if this leads to higher power usage.

Chapter 6 summarizes this thesis and discusses further research avenues.

Chapter 2

Related Work

In this section, we discuss the approaches for performance analysis of execution time and energy performance. We are interested in studies that relate multicore performance with off-chip resources such as memory or I/O. We discuss methods that address two types of analysis:

1. *Performance understanding* – We focus on accounting the parallel performance loss in multicore systems and understanding its causes. This allows performance understanding of parallel programs.

2. *Performance prediction* – The objective is to model the performance of a program on a target system, as a factor of number of cores, memory and I/O performance, under different system configurations.

First we discuss the related work in the area of parallelism performance, followed by approaches for analysis of power-energy performance. Lastly we summarize the limitations of the related work and highlight the differences to our approach.

2.1 Parallelism Performance Approaches

We start by discussing the theory of parallelism models, because they are the closest to our approach, followed by runtime overheads studies, focusing on the memory contention among cores. Next we discuss general analytical models and empirical methods, and then we cover measurement approaches for parallelism performance.

2.1.1 Theory of Parallelism

The theory of parallelism models rely on quantifying the parallelism of a program as the average number of work units that can be performed per unit time. The common assumption for these models is that they do not explicitly account for any type of runtime overheads, and therefore, all work is considered useful work. Nevertheless, these models are widely used in scheduling parallel tasks.

In the theory of parallelism models, a parallel program is represented as an directed acyclic graph (DAG) [19, 34, 94]. Each vertex in the DAG represents a subtask of the problem, with each subtask denoting a unit of sequential work, each with possibly different service time. The subtasks have precedence requirements that represent the data dependency of the problem. An arc in the DAG from an subtask to another means that the former must complete before the latter can begin execution. The critical path of this DAG denotes the longest sequence of serial calculations. If the service demand of the critical path is summed up, the result T_{cp} is the lower bound of the time required to finish the program:

$$T \geq T_{cp}$$

The sum of all the service demands of the DAG's vertexes denote the total work-

load of the program, which in turn is the upper bound of the time required to complete the program using one processor:

$$T \leq T(1).$$

Eager et al. [34] provide a simple model for the speedup of an application and the efficiency of executing it. The model relies on the average program parallelism π as the single parameter required to estimate speedup.

A notable result given by Eager et al. is that, given an unbounded number of processors, and in the absence of runtime overheads, the maximum speedup achievable when there is no bounds on the number of processors, $S(\infty)$, is equal with the average program parallelism:

$$S(\infty) = \pi(\infty) \tag{2.1}$$

The model assumes a work-conserving scheduling discipline, in which the processors will not stay idle if there is work that can be executed. Using this assumption, by knowing only the inherent parallelism π and n, the model establishes the lower S_{LB} and upper bounds S_{UB} of speedup:

$$S_{LB}(n, \pi) = \frac{n \cdot \pi}{n + \pi - 1} \tag{2.2}$$

$$S_{UB}(n, \pi) = min(n, \pi) \tag{2.3}$$

Both bounds are reachable.

If, in addition to π, the sequential fraction f is know, then the upper bound of the speedup can be strengthened to

$$S_{UB}(n, \pi, f) = min(\frac{n}{1 + (n-1)f}, \pi). \tag{2.4}$$

Eager et al. [34] argues that if it uses a estimate of the speedup located between the bounds, then this estimate $\hat{S}(n)$ is at most 34% away from the real speedup. The speedup estimate is given by the following formula:

$$\hat{S}(n,\pi) = \frac{2 \cdot S_{UB} \cdot S_{LB}}{S_{UB} + S_{LB}} = \frac{2 \cdot min(n,\pi)\dfrac{n \cdot \pi}{n+\pi-1}}{min(n,\pi) + \dfrac{n\pi}{n+\pi-1}}. \tag{2.5}$$

Downey [31] extends Eager's model to include the variance of the degree of parallelism $Var(\pi)$. For this, the model proposes two hypothetical parallelism profiles, one corresponding to a low-variance program and the other to a high-variance program. The paper then derives two speedup models, one for each type profile.

The parameters of the low-variance profile are chosen such that $\pi(t) = \pi$ for all but some fraction of the duration σ, with $0 \leq \sigma \leq 1$. The reminder σ fraction of the program, is evenly divided between a sequential part ($\pi(t) = 1$) and a high parallelism part, where $\pi(t) = 2\pi - 1$. With these carefully chosen values, the variance in parallelism of this program $Var(\pi)$ is thus

$$Var(\pi) = \frac{\sigma}{2} \cdot (\pi - 1)^2 + \frac{\sigma}{2} \cdot (2\pi - 1 - \pi)^2 = \sigma(\pi - 1)^2 \tag{2.6}$$

where the first term of the equation accounts for the variance in the sequential ($\pi(t) = 1$) region of the program, and the second term accounts for the variance in the high parallelism ($\pi(t) = 2\pi - 1$) part of the program.

The profile for the high-variance program has a sequential component ($\pi(t) = 1$) of duration σ and a parallel component where the parallelism $\pi(t) = \pi + \pi\sigma - \sigma$ of duration one. With these carefully chosen values, the variance in parallelism

$Var(\pi)$ is

$$Var(\pi) = \frac{\sigma(\pi-1)^2 + 1 \cdot (\pi - \pi + \pi\sigma - \sigma)^2}{1+\sigma} = \sigma(\pi-1)^2. \tag{2.7}$$

Since for both types of program, the variation of parallelism $Var(\pi) = \sigma(\pi-1)^2$, the paper argues that the semantic of σ is thus the square of the coefficient of variation, CV^2. Since $CV = \frac{\sqrt{Var(\pi)}}{\pi}$, the relation between σ and $Var(\pi)$ is:

$$\sigma = \frac{Var(\pi)}{\pi^2} \tag{2.8}$$

Using this semantic of σ, a program is considered low-variant if $\sigma < 1$, whereas if the program exhibits $\sigma \geq 1$ is considered high-variant.

The speedup S_{LVar} for a program with a low-variance DOP profile can be calculated as

$$S_{LVar}(n, \pi, \sigma < 1) = \begin{cases} \dfrac{\pi n}{\pi + \frac{\sigma}{2}(n-1)} & 1 \leq n < \pi \\ \dfrac{\pi n}{\sigma(\pi - \frac{1}{2}) + n(1 - \frac{\sigma}{2})} & \pi \leq n < 2\pi - 1 \\ \pi & 2\pi - 1 \leq n \end{cases} \tag{2.9}$$

while for the high-variance profile, the speedup S_{HVar} is:

$$S_{HVar}(n, \pi, \sigma \geq 1) = \begin{cases} \dfrac{\pi n(\sigma+1)}{\pi + \pi\sigma - \sigma + n\sigma} & 1 \leq n < \pi + \pi\sigma - \sigma \\ \pi & \pi + \pi\sigma - \sigma \leq n \end{cases} \tag{2.10}$$

$S_{HVar} = S_{LVar}$ when $\sigma = 1$.

Downey's model thus refines the speedup estimate that Eager et al. provides. When $\sigma = 0$, then S_{LVar} approaches S_{UB}, whereas when $\sigma \to \infty$, S_{HVar} approaches S_{LB}. However, the model does not address the cases where the degree of parallelism cannot be assigned clearly to low-variation or high-variation.

Limitations The major limitation of the theory of parallelism models is that they do not address how to determine the average parallelism of a program. Furthermore, determining π and $Var(\pi)$ using measurements of program execution limits the predictive value of the models, because it does not allow extrapolation with respect to machine size. A second limitation is that they do not address the runtime overheads, and do not model the parallelism loss due to runtime overhead.

2.1.2 Runtime Overheads Studies

In this section we discuss the related work of modeling the runtime overheads in parallel programs. We first present a general approach for accounting runtime overheads, and then focus on studies of memory contention among cores, because this type of overhead is more costly in current multicore architectures. We conclude this section by summarizing the shortcomings of current approaches for studying overheads in parallel programs.

When a program is run on a parallel machine, there will be runtime overheads that will impact the performance of the program. There is a multitude of factors that cause this degradation of performance, and often the performance analysis models do not explicitly account for them. For example, a common assumption in parallelism models is that the load balancing is perfect [8, 31, 34, 51, 61, 101]. However, load imbalances are widely encountered during program execution both as an unintended consequence of task partitioning [40] or deliberately due to the scheduler favoring other performance criteria besides load balancing, such as cache or memory locality among others [103].

Due to the large number of factors affecting the runtime overheads, it is intractable to model the effect of each of these factors. Therefore studying the impact of runtime overhead on performance involves first characterizing and classifying the overheads and second, modeling the effects only of those that are

deemed significant for performance.

One characterization of the overhead is attempted by Mark Bull in [21]. Bull's scheme classifies the temporal and spatial overheads in categories that are complete, meaningful and orthogonal. The overhead is defined as the difference between the *observed performance* and the theoretical *best performance* of a program running on n processors. The model defines the best performance, $T_{ideal}(n)$, as:

$$T_{ideal}(n) = \frac{T(1)}{n} \tag{2.11}$$

where $T(1)$ is the total work of the program, without considering any overhead. This definition is similar with the theory of parallelism execution time of an embarrassingly parallel program, in the absence of any runtime overhead. However, Bull's classification introduces several categories of parallel overhead, and extends the theory of parallelism model to account for runtime overheads. If $O_i(n)$ represents the overhead of category i incurred when running on n processors cores, then

$$T(n) = T_{ideal}(n) + \sum_i O_i(n). \tag{2.12}$$

Bull's complete categorization is presented in figure 2.1. The classes at the top of the temporal overhead classification scheme are:

1. *Control of parallelism* overheads which account for the runtime of the additional code executed to manage the parallel structures.

2. *Information movement* overheads are associated with data transfer between memory and processors and among the processors.

3. *Additional computations* overheads are required to expose the parallelism of the program to the threads or processes involved.

4. *Critical path* overheads result from the critical path being longer than the ideal.

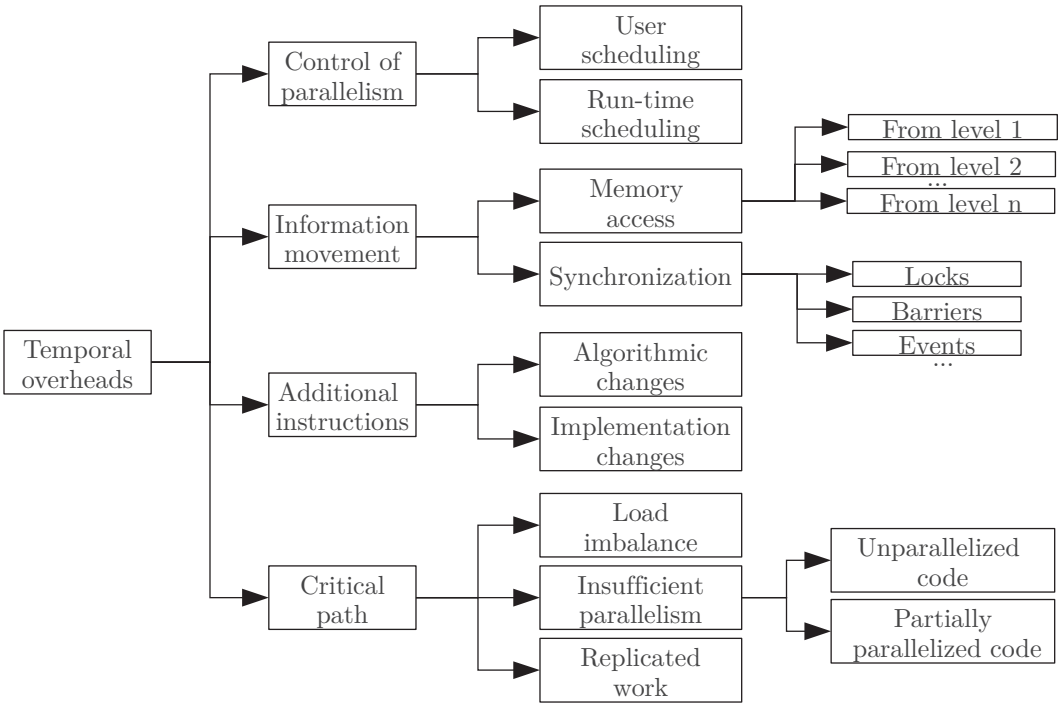


Figure 2.1: Overheads of a parallel program

The *control of parallelism* overhead comprises of *user scheduling*, which is time required for the application-level scheduling and task partitioning, and *run-time system* overhead, which accounts for the thread and process management.

In the *information movement* class, data accesses overhead refers to the time required to *access the data* from memory. This category is further divided based on the memory hierarchy and includes the transfer of data from other systems via network. *Synchronization overheads* are subdivided into the type of synchronization structures involved. It should be noted that this synchronization overhead does not include the time spent waiting at the synchronization points, because this time is included in the critical path overhead.

Additional computations might be needed due to *algorithmic changes* that are required to expose the parallelism of the problem or due to *implementation changes*

which are not explicitly required by the problem or by the parallel algorithm.

The overhead related to the *critical path* is caused by imperfect parallelization of the program. The main cause of this overhead is that optimal scheduling of the tasks is an NP-complete problem[33]. Specifically, given a set J of jobs where job j_i has service time l_i and a number of processors n, finding the minimum possible time required to schedule all jobs in J on n processors such that none overlap is an NP-complete problem. Since an efficient solution cannot be found quickly, the most common solution is to admit load imbalances and replicated work regions in the scheduling solution. *Load imbalance* denotes some processors to be idle even when there are enough parallel tasks, because they are asymmetrical in service time. Another component of the critical path overhead is the time-cost of *replicated work*: some pieces of the program are executed by more than one processor, even though it is only necessary to execute them once. The last component of the critical path overhead is the insufficient parallelism which denotes that at the current stage of the program there are not enough tasks for all the processors.

The major limitation of this classification is that some of the overheads are not easily quantifiable. For example, measuring the additional computations required by algorithmic or implementation changes is challenging. However, this type of classification stands at the base of the automatic overhead analysis done by tools like Ovaltine [14], ompP [40], Scal-Tool [106], and Scalea [113].

2.1.3 Memory Contention in Multicore Systems

As technological trends suggest that the increase in number of cores cannot be matched by the increase in memory bandwidth and speed [98], the memory contention problem [89] receives renewed attention.

Contention for shared resources in multicore systems has received significant attention in the research community. In general, studies of off-chip resource con-

tention fall in two main directions: reducing off-chip memory accesses [25, 37, 50, 73, 91, 95, 124], and improving the performance of off-chip requests [63, 74, 88, 90]. In reducing off-chip memory accesses, a major target for optimization is the last-level of cache memory. Partitioning of shared caches has been proposed as a technique to reduce the number of cache misses [73, 91, 95, 108]. Utility-based cache partitioning [95] uses specialized hardware to determine the miss rate of co-scheduled parallel programs, and partitions the available cache memory to reduce the overall miss rate. But, in software-based cache partitioning [25], operating systems page coloring is used to map the physical memory requests of a program to a reserved part of the cache. In cache-aware applications, co-scheduling is exploited to optimize cache miss fairness among different programs [37] or overall system performance [124]. Herdrich et al. [50] proposes throttling the speed of the cores to generate an imbalanced number of cache misses for relieving contention in applications with different memory access intensity. There are many approaches to improve the performance of off-chip requests. Memory bandwidth partitioning [57, 63, 74, 88, 90] has been proposed for optimizing different performance criteria. Kim et al. propose ATLAS [63], a memory controller scheduler that prioritizes threads with least-attained service levels to improve the overall performance of co-scheduled threads. The Fair Queue Memory System [90] ensures that co-scheduled threads receive a predetermined fraction of the memory bandwidth regardless of other threads memory requirements. Liu et al. [74] studied and modeled the interaction between cache and bandwidth partitioning with the goal of optimizing the overall performance of co-scheduled threads.

While there are many studies to reduce memory contention, there are few general models that directly link the performance of parallel applications to resource contention, number of active cores, problem size and the patterns of memory access.

Liu et al. propose a general analytical model for understanding the effect of bandwidth fraction on individual thread performance [74]. Based on the cache miss ratio, their model determines the CPI performance of co-scheduled threads, and the slowdown of co-scheduling groups of threads relative to scheduling each thread individually. However, it is unclear how the fraction of last-level cache misses is determined when different number of threads are scheduled together. Their model also does not explore changes in problem size, in particular large problem size that is typical for parallel programs.

Using high performance applications, Hood et al. [53] propose a model to determine the performance impact of shared-resource contention such as cache, bus, memory controllers and processor interconnects. Their differential performance analysis approach measures the performance for different configuration scenarios. However, their approach does not apply to predictive performance analysis, nor does it take in consideration the problem size and burstiness patterns.

Sancho et al. [99] study the relationship between memory bandwidth and performance of parallel programs when the number of memory channels of each memory controller is changed. Their approach is based on measurements of memory bandwidth and parallel processing rate, and shows that increasing the number of cores exerts higher memory demand and has diminishing results on performance due to memory bandwidth saturation. Their focus is to understand which configuration offers the best memory bandwidth.

Xie and Loh [121] propose to classify multithreaded applications using its overall behavior on shared last-level cache. Applications are divided into four categories based on its request intensity, and how it interacts with other applications. However, their approach does not cover the impact of the number of cores on which these applications are scheduled, or understanding of the relative size between the last-level cache and the working set of the applications.

Limitations Recent studies about on-chip and off-chip memory contention are proposed in conjunction with software and hardware improvements to available multicore systems. Therefore, most studies explore a small parameter space, and thus are hard to generalize to other multicore systems. In particular, the effect of active number of cores and problem size on memory contention is not addressed by most of the related work. Even when the number of cores is modeled [74], the main objective is not to relate to overall parallelism performance, but rather with proposed hardware and software improvements. Finally, another limitation is that most studies involve simulations either for validation or for experimental observations [63]. Due to the prohibitive cost of simulating large problem sizes, these approaches are not suitable for studying the performance of large parallel programs.

2.1.4 General Analytical and Empirical Models

Recent performance analysis methods for multicore systems recognize that the performance of large parallel programs often depends on hundreds of parameters including programming model, memory architecture, problem size, partition size among others. Thus, there is a pronounced shift towards performance analysis methods that emphasize the ease of determining and applying an analytical model that predicts the performance of the system [111].

In this section, we discuss two types of predictive models. General analytical models derive an equation of the parallelism performance, as a factor number of active cores and various software and hardware metrics. In contrast, empirical models fit an equation over data acquired during several runs on the program using different software and hardware configurations.

General Analytical Models Among the general models that predict the performance of a program, Amdahl's law [8] is the most widely known. Amdahl's law derives the speedup of a program as a factor of the sequential fraction f and number of processors n:

$$S(n) = \frac{1}{f + \frac{1-f}{n}} \tag{2.13}$$

and from this relation to derive the upper bound of S as

$$S(\infty) = \frac{1}{f}. \tag{2.14}$$

Using this equation, the model can be used to predict the overall speedup gained by improving only one section of the program [92].

Amdahl's model shows that the sequential fraction of the application impairs performance to a large degree, as even minuscule sequential fractions can compromise large speedups. In order to gain a high speedup, the sequential fraction must be minimized. Moreover, equation 2.14 shows that the speedup obtained in a program that has a sequential fraction f is bounded by $1/f$. Gustafson [47] has shown that the sequential fraction depends on the size of the problem, thus showing that scalability increases together with the problem size.

Another class of general models combines the theory of parallelism with sequential fraction, to derive equations of the parallel speedup [31, 32, 34, 101]. These models are discussed in under the theory of parallelism approaches.

In general, Amdahl's model and its extensions are easy to apply, but have a limited value for understanding the performance loss in complex applications. Using sequential fraction makes the models hard to apply, because f is inconsistent across hardware and software platforms. Furthermore, simply dividing the work of a program into purely sequential and embarrassingly parallel is too simplistic

to model the data dependency in a modern parallel program. Lastly, Amdahl's model abstracts various types of overheads under a single parameter, the sequential fraction. Understanding the parallelism loss due to the various overheads becomes difficult under this model.

Empirical Models Empirical methods use a number of runs of the programs, called baseline runs, to determine a mathematical function that fits an observed performance [15, 28, 41, 56, 105, 107]. This function then allows various predictions about the system. Generally, these models use multiple linear regression [15, 105], machine learning [41] or neural networks [105] using data measured in various parameter configurations to relate the measured performance metrics of the program to the changes in the configuration parameters. These models have good accuracy and generally low intrusiveness, and therefore they are used for decision making in ACTOR runtime system [27].

Barnes et al. [15] explore regression to predict the scalability of distributed-memory applications. They use program executions a small set of processors to predict the behavior on enlarged sets.

The approach is to run the same program on a set of q processors, where $q \in \{2, 4, \ldots, p_0\}$ and $p_0 < p$. The input variables of the program $(x_1, x_2, \ldots, x_k)$ are varied on each run and the execution time of the program for each of these runs is considered as a function of the input variable and number of processors. Thus, the model develops a predictor $\hat{T}$ of the execution time T:

$$\hat{T} = F(x_1, x_2, \ldots, x_k, q). \tag{2.15}$$

The goal of the model is now to determine the predictor $\hat{T}$ in such a way that they

minimize the relative error $E = \frac{|T-\hat{T}|}{T}$. Equation 2.15 can be developed as

$$\begin{aligned} log_2(T) &= log_2(F(x_1, x_2, \ldots, x_k, q)) + error \\ &= \beta_0 + \beta_1 log_2(x_1) + \beta_2 log_2(x_2) + \ldots + \beta_k log_2(x_k) + \beta_q log_2(q) + error \end{aligned} \tag{2.16}$$

The most important part of this equation is the term $\beta_q log_2(q)$ which models the impact of the number of processors. This term is further expanded as $g(q)$:

$$g(q) = \beta_q log_2(q) = \gamma_0 + \gamma_1 log_2(q) + \gamma_2 log_2^2(q). \tag{2.17}$$

The model considers two versions of this equations: the quadratic coefficient γ_2 as zero (in which case the function $g(q)$ is a simple linear function) and as non-zero.

A refinement of this model is to separate the communication and computation times, as the authors suspect that they scale differently. They used instrumented programs to determine for each run what is the communication time and what is the computation time, and then they apply the regression model for each of these two times.

They used the training runs to estimate the β and γ parameters that minimized the relative error. The number of training runs is between 10-30 per input variables. The training runs are conducted on a number of processors of $p_0 = p/8$, $p_0 = p/4$ and $p_0 = p/2$.

In general, empirical models have very good accuracy. The validation of the model against measurements on the same number of processors shows that the accuracy for predicting up to 1024 processors is 13%.

Limitations The main limitation of general analytical models is caused by their simplifying assumptions. Some of the assumptions may reduce the accuracy of the model. For example, the use of the sequential fraction f in Amdahl's law and its

extensions compounds many important overheads that affect the parallelism of the program. Moreover, the sequential fraction f lacks consistency across many software and hardware platforms. As such, determining f from a run on a particular configuration might not be suitable in applying the model for other configurations. Furthermore, the sequential fraction is a simplistic model of the data dependency in the program, because it considers the parallelism either sequential or embarrassingly parallel. In reality, the parallelism can take any value between one and the maximum parallelism, as shown by the theory of parallelism models.

The disadvantage of empirical approaches is that it does not allow qualitative insights into the problem, but merely provides a quantitative estimate. These models do not permit the separation of the different factors that affect the scalability, for example, the impact of workload changing and of changing the core frequency or memory topology. Another disadvantage is that using regression requires a large number of training runs to get an accurate prediction. Usually the validation is done only for predictions within the same order of magnitude as the training runs. For example, in Barnes et al. the maximum increase in p is eight times p_0. Therefore, it is unclear how accurate is the model if the prediction is done for a number of processors orders of magnitude larger than the number of processors used for the training runs.

In summary, the main limitations of general analytical models are their limited predictive ability and their lack of accuracy if the model assumptions do not match the behavior of real program executions. In contrast, empirical methods have better predictive capabilities, but they are limited in providing performance understanding for different types of parallelism loss, and require a large number of baseline runs to train the models.

2.1.5 Measurement and Instrumentation

Methods relying on program instrumentation are used to obtain detailed insight on the performance of a program. In general, the approach to instrument a parallel program includes source code instrumentation or binary code instrumentation. Source code instrumentation methods include OPARI to instrument OpenMP programs, PMPI for MPI programs, or GCC profiling support for general C, C++ and Fortran codes, among many others [39, 46, 64, 65, 100]. Among binary code instrumentation tools, PIN [77] is the most widely used.

In general, models relying on instrumentation, have the best accuracy, but incur a large cost of applying them. Instrumented executions are typically substantially slower, because of the extra code injected in the application. Furthermore, it is known that instrumentation is intrusive and often prevents some types of optimizations. For example, OPARI which is at the heart of KOJAK, TAU, Scalasca [46] and ompP [39] tools, prevents the usage of implicit barriers, which in turn prevents the OpenMP NOWAIT clause, thus forcing the threads to perform an additional synchronization operation. Instrumentation may slow down the program or interfere with cache sensitive areas, and therefore increase the overhead of the parallel programs. Some vendors provide highly optimized versions of popular parallel kernels (such as BLAS or LAPACK) which come directly compiled as libraries. Without access to the source code of such products, prediction methods relying on instrumentation may be inapplicable.

Another disadvantage of measurement and instrumentation methods is the cost of logging the events and processing the logs. High resolution sampling of the executions might create logs which are substantial in size. Furthermore, writing the logged information to memory or to the disk might perturb the execution of the program. For example, writing the logged information into memory during

the execution of cache-sensitive compute phases might perturb the cache balance, inducing an additional overhead, compared to non-instrumented executions.

However, the major disadvantage of instrumentation methods is their lack of generality across different software and hardware platforms. Source code instrumentation tools target a limited number of programming languages and models, and can only be applied when the source code of the program is available. Binary code instrumentation tools can only be applied for a specific hardware platform. For example, PIN only supports Intel and AMD architectures.

2.2 Power and Energy Studies

The focus of our power and energy models is programs with significant compute, memory or I/O requirements. Therefore, the discussion of related work on power and energy models is conducted with respect of techniques for analyzing the power performance of compute, memory and I/O systems.

The study of power and energy usage in parallel system has received considerable attention in the literature. Before the shift to multicore, the power models addressed the impact of the core frequency f on overall power and energy consumption. After the clock frequencies have plateaued, power models shifted to understanding the impact of increasing the number of cores n on power and energy cost.

We structure the related work on approaches for power and energy profiling can into three parts: (i) modeling and simulation approaches, (ii) measurement and empirical methods and (iii) energy-efficiency studies.

2.2.1 Analytical Modeling and Simulation Approaches

Rapid increases in clock frequencies and number of transistors per die have dramatically increased the complexity of the processors, and their power consumption. Thus, power consumption and dissipation have become key concerns in many multicore systems.

In general, due to the complexity of processor microarchitectures, completely analytical models for power and energy usage are difficult to employ. Instead, simulations of CPU microarchitecture are used to derive the power and energy cost either at microprocessor component level, or at chip level [24, 78, 118].

Most simulation approaches employ the general analytical model for power consumption of a CMOS circuit:

$$P = N_{sw} \cdot C_l \cdot V_{dd}^2 \cdot f \tag{2.18}$$

where N_{sw} is the switching activity performed by the CMOS circuit, C_l is the load capacitance, V_{dd} is the operating voltage and f is the clock frequency of the digital circuit [71]. The general power equation suggests that power consumption depends quadratically on voltage and linearly on frequency. However, because processor chips are composed of many distinct CMOS circuits operating at different frequencies and voltages, the relationship between total power required by a chip and its operating frequency is more complicated. Nevertheless, the general power equation is the basic model behind power and energy simulators.

In general, simulators for power and energy use a cycle-accurate execution model of a processor. All the processor components, including functional units, control units, internal data-paths and internal buses among others, are completely specified. During the cycle-by-cycle execution, the simulator analyzes all the com-

ponents that are used from the processor and cumulates their power usage, as modeled by the general power equation [12, 118, 122].

SimplePower [122] is a cycle accurate, execution driven register transfer level (RTL) simulator. It simulates the integer subset of the instruction set of Simplescalar [12]. Its architecture is based on a five-stage pipelined data-path, consisting of instruction fetch, instruction decode, execution, memory access and write-back stages. At each clock cycle, it simulates the execution of all active instructions and activates the corresponding functional units. For each activated functional unit, a RTL interface is used to derive the power consumption in a technology-dependent way. For each fabrication technology supported, the simulator includes a table of capacitances and voltages. SimplePower includes an instruction and data cache simulator and a memory bus interconnect. To simulate a complete processor-memory system, SimplePower is usually coupled with a memory simulator, such as CACTI [118].

Limitations. There are three major limitations of using simulators for power and energy analysis. First, cycle accurate simulators have a large execution cost and simulating large problem sizes is currently untractable. This is particularly problematic because large parallel programs are common workloads in current multicore systems. Second, simulators do not scale well in terms of the number of simulated cores. Third, simulators are often used to isolate the performance of specific components. For example, a common assumption when simulating a processor is that the last level cache is infinite. This assumption is reasonable in the context of understanding the performance of the processor alone, but is invalid for understanding the performance of the entire system.

2.2.2 Measurement and Empirical Approaches

When the level of analysis is chip-wide or system-wide, the power and energy cost depends on a multitude of factors, including number of functional components inside the cores, number of cores, size, types and number of caches, buses, memory systems, fabrication technology among many others. For such analysis, measurement and empirical methods are attractive approaches, because it allows easy understanding and predictive analysis.

In general, measurement analysis approaches use two types of methods for reading the power values: *external instrumentation* and *internal instrumentation* [17, 22, 58, 68, 71].

In external instrumentation, external power measurement instruments are connected to the power supply of different components of the system [22, 58]. The devices record directly the power and energy use. This methods have the advantage that it is suitable for many types of devices and typically it is very accurate (the accuracy depends on the measurement instrument). The main disadvantage is that the instrument needs to be physically connected to the component that is being profiled. In many cases, this is very difficult, for example in system-on-a-chip, because many components are physically packaged as one device. However, for system-wide analysis, external instrumentation typically presents the most accurate method.

In contrast, internal measurement approaches use software values output by the system's sensors. These approaches are further divided into hardware-independent methods and hardware dependent methods. Advanced Configuration and Power Interface (ACPI) [1] is a hardware-independent standard for power monitoring and reporting. ACPI is implemented by many components in modern systems and helps control the power states and power consumption by the operating system.

For example, in battery-powered devices running Linux, ACPI reports the voltage and current drawn from the battery. ACPI forms the basic monitoring method for devices running Linux, Mac OS X or Android [22, 58]. However, the main limitation of ACPI monitoring is the resolution and its accuracy. ACPI reports the values of voltage and current using the `procfs` pseudo-filesystem and the time between two updates is in the order of seconds. As such, its accuracy is reduced for workloads which are very dynamic. Furthermore, the values of power consumption are drawn at system-wide level. Hardware dependent methods rely on platform-specific sensors and monitoring tools [58]. Typically they have better resolutions and allow a breakdown of power consumption per component.

Because most power-measurement approaches report the power at system-wide level, many studies use empirical models that separate the total power into components [30, 58, 69, 70, 104]. Workloads designed to stress particular components of the system are used to perform a differential analysis between the system without component stress and system without component stress. In general, these empirical studies use baseline runs of different programs on a target system, and collect power and energy readings in configurations that activate different components of the system.

Bertran et al. produce an empirical decomposable model of power and energy performance of a multicore processor, based on correlating power usage with hardware events counters [17]. Their approach is to first define modeling inputs, which are the power component activity ratios. Next, they define the training data which is generated using microbenchmarks and collect training data. Then, they model the power consumption for individual components.

The approach to determine the utilization of different processor components is to separate the architecture of the processor into in-order engine, memory subsystem and out-of-order engine. For each of these components, they compute a

formula of power activity ratio AR based on a ratio of the value of hardware events counters. For a single core, processor, their model for power consumption of a single core is:

$$P_{total} = (\sum_{i=1}^{\#comp.} AR_i \cdot P_i) + P_{static}$$

where AR_i is the activity ratio (i.e. utilization) of component i, P_i is the power consumption of the component under full load, and P_static is the total static power of the system. P_i is derived from the training data using multiple linear regressions. They extend the power model for single core to multiple cores, considering that static power does not change when the number of active cores are changed.

$$P_{total} = \sum_{j=1}^{\#cores} ((\sum_{i=1}^{\#comp.} AR_{ij} \cdot P_i) + P_{static})$$

The model is validated against measurements conducted on configurations different from the training runs and the accuracy is showed to be under 3%.

Models that focus on energy performance are significantly more complex than power models. This is because they require a model of both power and execution time. In general, the studies focus on embedded systems, where execution time is modeled using a bottleneck analysis [26, 60, 71].

Liang et al. create a predictive empirical model for energy usage of single-core ARM systems with DVFS support. Their model considers that the execution of instructions includes two portions:

1. Time spent in ideal CPU operations, such as integer instructions, floating point instructions and control flow operations.

2. Time spent in external memory accesses, which is determined by the number of cache misses.

The model considers that number of CPU cycles incurred by the program, consists

of memory cycles and ideal cycles: $N = N_{ideal} + N_{mem}$. The time for the ideal CPU operations is $\frac{N_{ideal}}{f}$ and the time required for the external memory operations to complete is:

$$T_{mem} = \frac{N_{ideal}}{f} + \frac{N_{mem}}{f}$$

Because the execution of the instruction overlaps waiting for data from memory, when the memory is the bottleneck, $T_{CPU} = T_{mem}$. For the other case, $T_{CPU} = T_{ideal}$. In order to model the memory access characteristic of the program, the model defines memory access rate, MAR:

$$MAR = \frac{\text{Instruction Cache Misses} + \text{Data Cache Misses}}{\text{Number of Instructions Executed}} \tag{2.19}$$

and derives the equation for the total execution time of a task as

$$T_{TASK} = T_{CPU} + MAR \cdot T_{BUS} \tag{2.20}$$

where T_{BUS} is the bus time, and is related to the bus frequency. They regress the inputs of the model using data acquired using a series of training runs. Using this model, the authors derive a frequency, called *critical speed* that achieves the minimum energy usage. They further derive an empirical equation for a particular platform, which predicts the critical speed as a function of MAR. Using this equation, they predict the frequency that minimizes the energy usage of a program for a single core system.

Limitations. While empirical methods usually have good prediction accuracy, they often have little value for performance understanding, especially in the context of parallel applications on multicore systems. For example, Liang et al. only model single core systems, and do not address the effect of problem size on per-

formance. Furthermore, the model requires training runs that cover all available frequencies of the system, which constitute a significant effort for systems with large number of DVFS steps.

2.2.3 Energy-Proportionality in Multicore Systems

In this section, we discuss the related work on energy-proportionality and energy-efficiency of multicore systems. Energy-proportionality refers to the desirable property of a system that consumes energy proportional to its useful work output [16]. Computing systems often have poor energy-proportionality because most commodity hardware components have a significant idle power usage, and often they exhibit sublinear increase in power and energy usage with an increase in utilization.

An important aspect that directly impacts the energy efficiency is the problem of selecting the optimal number of cores in a multicore system. This question has been addressed from the perspective of selecting the optimal performance of area-equivalent cores that, when replicated across the entire die, offers the best system-wide throughput. Many studies have addressed the dichotomy of using the transistors budget of a chip to create either few powerful cores (termed *brawny cores*), or many less powerful cores (termed *wimpy cores*) [29, 45, 51, 52, 76, 93]. In general, their conclusion is that wimpy cores may offer better system-wide throughput than the area-equivalent high performance cores, if two considerations are met. First, the workload is stationary and has enough parallelism to sustain execution on many cores [51]. Second, the relative performance between wimpy to brawny nodes does not impact the overall cluster cost, programability and schedulability of the parallel tasks [52]. By obtaining better system-wide throughput with the same energy costs, wimpy cores are shown to be more energy-proportional than the area-equivalent brawny cores, for programs with relatively

high degrees of parallelism.

Hill and Marty [51] propose a modeling study that extends Amdahl's Law to heterogeneous multicore systems. Their conclusion is that workloads with a realistic degrees of parallelism profile would achieve better performance using heterogeneous cores, rather than a system with fully wimpy cores or fully brawny cores. Systems with heterogeneous cores include ARM big.LITTLE which combines two powerful ARM Cortex-A15 cores with three power-efficient ARM Cortex-A7 cores, and Bahurupi [93] which provisions the cores with the ability to morph into coalitions with high execution rate.

A more recent focus on power-aware computing recognizes that off-chip resources are becoming key in exploiting energy-proportional executions. For example, the energy used by memory is a growing concern, with 30–57% of the energy spent by a server being attributed to the DDR3 DRAM memory chips [123]. Even worse is that mainstream memory chips such as DDR3 have poor energy-proportionality: a DDR3 memory subsystem used 20% will consume almost half the power used when fully loaded [81]. As such, recent work has proposed using low power DDR (LPDDR), that is typically used in embedded devices, as the memory banks of future multicore servers [81, 123]. Such servers will typically have lower peak bandwidth, but negligible impact on web-hosting workloads and a reduction by 3–5 times of memory power. Similar concerns have lead to designs of energy-efficient processing systems in networked [9] or database systems [66] where key for efficiency is to improve the energy-proportionality of I/O devices.

As more hardware devices are becoming energy-proportional, the perspective of low-power but high-performance computing is becoming a reality. However, a reduction of power consumption at all costs is not always possible or beneficial. For example, during an application execution there might be small periods of computational bursts that are better to be performed without deferment either

because of a deadline [72] or because it is more energy-efficient to execute them on a high-power device [96, 97]. As such, an increasing concern in energy-proportional computing is to understand the relationship between power and performance, such that a reduction of energy consumption is performed through dynamic power allocation to the right hardware component at the right time [87].

Limitations. The main limitations on the related work on power-proportionality is their focus on on-chip resources only. Many studies focus only on the microarhitectural level [29, 51, 93] or at most address the on-chip cache [76]. As such, the impact of off-chip resources such as memory and I/O on the performance of brawny or wimpy cores is unknown. Our work shows that balancing the cores and off-chip resources improves energy-proportionality even more than focusing on on-chip resources only, and leads to lower energy consumption but sometimes with a higher power cost.

2.3 Summary

Current performance analysis approaches can be divided into methods for performance prediction and methods for performance understanding. Furthermore, the methods are compared based on three key design trade-offs: difficulty of use, intrusiveness and inaccuracy. Table 2.1 summarizes the type of analysis and the limitations of commonly used approaches.

Approach	Analysis Type		Limitations		
	Prediction	Understanding	Difficult to Use	Intrusive	Inaccurate
Analytical Models	Yes	Yes	No	No	Yes
Empirical Methods	Yes	No	Yes	No	No
Instrumentation Methods	No	Yes	Yes	Yes	No
Our approach	Yes	Yes	No	No	No

Table 2.1: Limitations of commonly used performance analysis methods

General analytical models [8, 31, 34, 101] are easy to apply, but often the

simplifying assumptions reduce accuracy below what is useful for practical purposes. Instrumentation methods and trace-driven analysis [64, 65, 100] require modification of the program which might result in lower performance compared to the non-instrumented case. While they have good accuracy, these approaches are often intrusive. This leads to difficulty in generalizing them across programming languages and models. Recently, a shift in analysis methods recognize that the performance of large parallel programs depends on a multidimentional space of options and configuration parameters, including programming models, number of threads and processor cores, problem size, memory architecture, thread-to-core placement among others. Therefore, the ease of applying the model across this parameter space is becoming a crucial design criteria for performance analysis methods [28, 111]. Methods that rely on empirical data, such as regression based-approaches, neural networks and machine learning [15, 28, 41, 105] typically produce good accuracy but they require significant modeling effort or large volume of training data. Furthermore, they typically have very good prediction accuracy, but they do not address the topic of performance understanding.

Our approach is to create a general analytical model with inputs derived from measurements. Our general model breaks down the parallelism of a program into useful work and various types of overheads. Using observations derived from measurements, we model the data dependency and memory contention. The models of data dependency and memory contention use widely available metrics derived from the trace of the operating systems run-queue and from hardware events counter collected during a few baseline runs. Using this approach we create a model for understanding and predicting the performance of shared-memory programs on multicore systems. Our proposed model achieves good accuracy because it is based on observations derived from measurement analysis. Furthermore, our approach is easy to apply and has little intrusiveness.

Chapter 3

Proposed Analytical Models

This section presents our modeling approach for analyzing the parallelism and energy performance of parallel programs on multicore systems. First we describe the model overview and approach. Next we introduce the parallelism performance model with its definitions, the data dependency model, the memory contention model, and the I/O overhead model. This is followed by the power and energy model.

3.1 Overview and Approach

The objective of the general analytical models is to describe the execution time performance, as well as the power and energy usage of a program running on a target multicore system.

We propose a general parallelism model that predicts the execution time of a program on a target machine configuration. The approach starts by modeling the inherent parallelism of a program as the useful work. Next, during the execution of the program on a given machine configuration, the inherent parallelism maps onto exploited parallelism and parallelism loss due to runtime overheads.

The general model is implemented for shared-memory programs spanning key application domains such as high-performance computing, multimedia, financial computing and web-hosting. For these applications, we identify three key execution resources – cores, memory and I/O devices, and derive expressions for the parallelism loss due to data-dependency among threads, memory contention among cores, and I/O overhead due to the network I/O operations. Furthermore, we implement the models by linking the useful work and runtime overheads to three key execution resources in modern multicore systems: processor core resources (i.e. the number of cores and core frequencies), memory resources (i.e. memory bandwidth and topology), and I/O resources (i.e. device bandwidth and I/O operations latency).

The key technique used to model the parallelism loss is *bottleneck analysis*. During the execution of a program, there is a large degree of overlap between the servicing of useful work by the cores, of the memory requests by the memory subsystem, and of the I/O requests by the network device. The execution time of any phase of a program is dictated by the resource with the largest service time, which is the bottleneck device. The service time of all the other resources are completely overlapped with the service time of the bottleneck device. Furthermore, our targeted programs spanning HPC, multimedia, financial and datacenter domains, consists of multiple iterations of the same compute, memory and I/O execution phases. Thus, it suffices to model only one execution phase to infer the entire execution behavior.

The proposed model for power and energy performance predicts the energy usage as a factor of active number of cores and clock frequency of the cores. The energy model is build upon the parallelism performance model, using execution time predictions from the parallelism model and a static power characterization of the system. In contrast with the execution time model, the energy is not

determined as the maximum among cores, memory and I/O energy usage, but rather as the sum of the energy usage of each component.

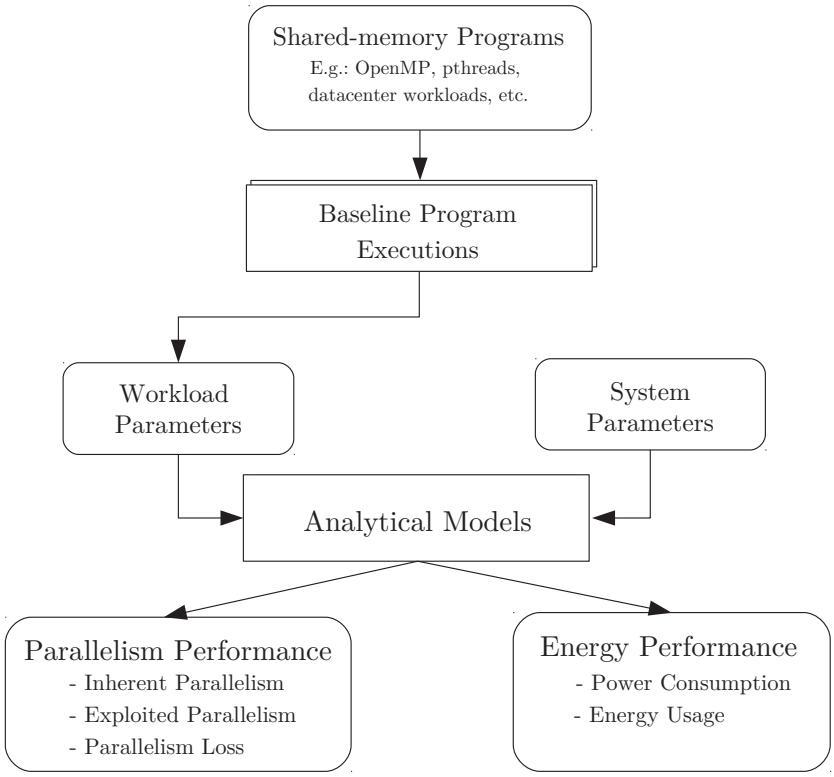


Figure 3.1: Approach for applying the model

The approach for applying the general model of parallelism is described in figure 3.1. Given a shared-memory program, we perform a set of baseline executions using a small number of runs. During these executions, we collect traces of software and hardware metrics and traces of power and energy use. We use the software and hardware traces as the inputs into the *parallelism model*. The objective of this model is to predict the inherent parallelism of a program, its exploited parallelism on a multicore systems and the parallelism loss due to data dependency and memory contention. The traces of power and energy are used in conjunction with the parallelism model by the *power and energy model* to predict the power and energy usage of the program.

The novelty behind our approach lies in bridging the abstract view of program

parallelism to the hardware parallelism of a modern multicore platform. While previous work has used metrics such as degree of parallelism [54, 101] and average parallelism [31, 32, 34] to express the performance of a program, they have not followed up with a model that reflects how such abstract metrics are tied to what is measurable in the hardware of a real system. Our model shows *how* the software view of parallelism matches the hardware parallelism, and provides a practical method for inferring the application performance from hardware events metrics available on contemporary and emerging multicore systems.

Table 3.1 shows the notations used throughout the thesis.

3.2 General Parallelism Performance Model

The objective of the parallelism performance model is to link the inherent and exploited parallelism of a shared-memory program to hardware events that are directly observable or measurable. This allows the direct determination of the inherent and exploited parallelism, and of the parallelism loss due to parallel overheads.

In a software view of parallelism, a parallel program can be modeled as a directed acyclic graph where the nodes are *work units* and the edges represent the logical dependencies between work units [31, 32, 34, 54, 101]. In general, the study of program parallelism uses the number of work units per unit time as the *degree of parallelism* of the program. Different units of work can be used, depending on the *type of the parallelism* that is analyzed.

From problem to execution, we distinguish three types of parallelism:

1. *Ideal parallelism* of a problem denotes the average number of work units that can execute concurrently at problem level. At this level, the work units are problem-dependent. For example, considering the problem of sorting integer

Symbol	Description
General parameters	
m	Number of threads
n	Number of cores
f	Clock frequency
t	Time moment
T	Time interval
c	Total cycles incurred by program
w	Work cycles executed by program
a	Stall cycles due to contention
b	Stall cycles not due to contention
r_M	Last level cache misses
r_I	Data transferred by I/O device
Parallelism performance	
π	Inherent program parallelism
π'	Exploited program parallelism
π_d	Parallelism loss due to data dependency
π_μ	Parallelism loss due to memory contention
π_σ	Parallelism loss due to I/O operations
U	Useful work expressed as time units
M	Memory contention, as time units
C	Work expressed as time units
I	I/O waiting time
ω_μ	Memory contention factor
ω_σ	I/O overhead factor
Energy performance	
P	Average power consumption of a program
E	Total energy usage of a program
η	Energy proportionality factor

Table 3.1: Table of notations

numbers, the unit of work is swap operation between two numbers, and the parallelism of this problem is the average number of swap operation that can be performed per unit time. The ideal parallelism is determined by the problem size and the parallel algorithm, which dictates the dependencies between the work units.

2. *Inherent parallelism* of a program [54] denotes the average number of work units in a program that can execute concurrently per unit time, at a logical level, in

the absence of any execution constraints. The parallel problem is transformed into a parallel program using a parallel programming model. Depending on the programming model, the ideal parallelism work units are mapped to other work units called *parallel tasks*. In *implicit parallelism* models, the compiler or runtime system automatically identifies the parallel work units and maps them to parallel tasks. For example, Go programming language [2] is an implicit parallelism programming language that maps each function to a parallel task called *goroutine*. In contrast, in *explicit parallelism models*, the programmer identifies, exposes and controls the parallelism, via programming language constructs. For example, C with pthreads is an explicit parallelism models, because the programmer manually assigns the work units to different threads, The effect of both implicit and explicit parallelism models is a constraint of the parallelism of the program, because the parallelism is upper-bounded by the number of parallel tasks generated by the programming models.

3. *Exploited parallelism* of a program denotes the number of processor cores that are executing *useful* work per unit time, averaged over the entire execution of the program. When a program is executed on a physical system, the threads of the programs are mapped to physical cores. The mapping between the threads or processes and the cores is controlled by a runtime scheduler. The effect of this mapping is that the number of physical cores constraints the exploited parallelism. Another effect is that there is runtime overhead associated with executing on a physical system, and therefore, not all work is considered useful.

From inherent to exploited parallelism, there is a parallelism loss caused by runtime overheads. Even without resource constraints, the three most common overheads in parallel programs are caused by memory contention among cores, communication and I/O operations [21]. Let π denote the inherent parallelism of

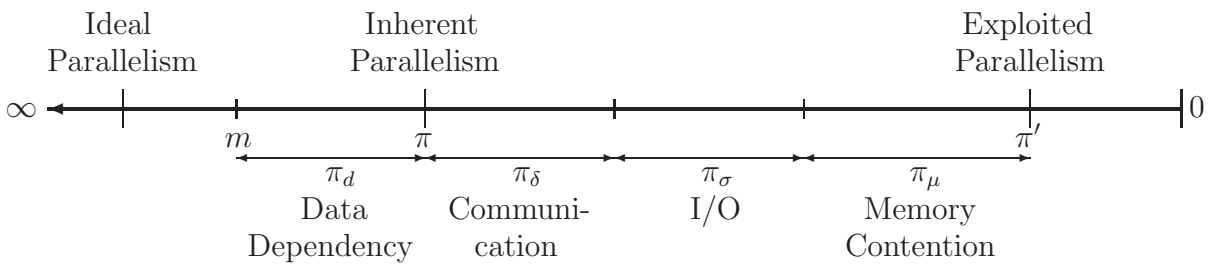


Figure 3.2: Breakdown of general program parallelism

a program partitioned into m parallel tasks and π' the exploited parallelism when executing on n cores. For the parallelism loss, let $\pi_d(m)$ denote the parallelism loss due to data dependency, π_μ due to memory contention, π_δ due to communication and π_σ due to I/O operations. Figure 3.2 shows the break-down of the parallelism of a program into inherent parallelism, exploited parallelism and parallelism loss.

This thesis focuses on *shared-memory programs* running on multicore systems such as Intel and AMD x64 or ARM Cortex family. Therefore, from the general model of parallelism, we do not consider the parallelism loss due to communication, because communication is an overhead specific to distributed memory. Furthermore, we focus on shared-memory programs where the I/O operations cover only *network I/O operations*, and not storage I/O. Therefore we only further model the parallelism loss associated with these overheads:

1. π_d – data dependency among the work units;

2. π_μ – memory contention among cores;

3. π_σ – network I/O overhead.

The breakdown of the parallelism and parallelism loss in a shared memory program is shown in figure 3.3. For shared-memory applications, our observation is that we can analyze the program parallelism directly at thread level. Even if some programming models may have different parallel work units at program level,

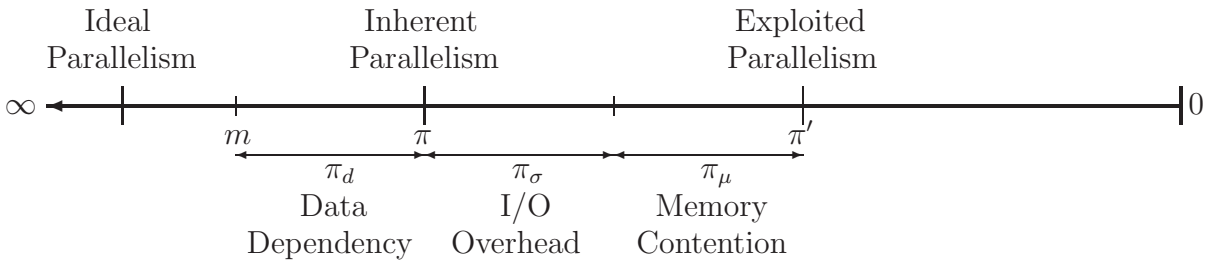


Figure 3.3: Breakdown of shared-memory parallelism

all shared-memory programming models are build on top of threads, and eventually all program-level parallel work units are mapped to threads. Furthermore, the choice of threads as units of work abstracts the differences between different programming languages, threading packages, operating systems and hardware platforms. However, the parallelism of a shared-memory program cannot be simply estimated by counting the number of threads, due to the data-dependency gaps in the execution time and due to runtime overheads.

Next we define the inherent and exploited parallelism of shared-memory programs, the useful work of a program, and the parallelism loss due to data-dependency and memory contention.

3.2.1 Inherent and Exploited Parallelism

At a logical level, if the number of execution resources available to a parallel program is unbounded, then the degree of parallelism of the application is dictated only be its inherent structure. To model the parallelism of a program, we extend the widely used concept of inherent parallelism [32, 34, 54, 101] to accommodate the effect of partitioning a shared-memory program into threads. Considering a shared-memory program with m threads, the program starts at time moment 0 and finishes at time moment T. Threads are *active* if they are executing work, or *suspended* if they are stopped at a synchronization point such as a mutex,

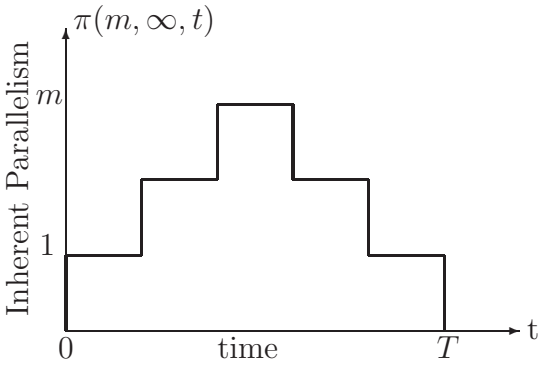


Figure 3.4: Degree of parallelism profile

semaphore or condition variable. Figure 3.4 shows the parallelism profile of a program. The parallelism of the program is always lower-bounded by one (there is always at least one thread active) and upper-bounded by m (when all threads are active). Let $\pi(m, \infty, t)$ be the degree of parallelism at time moment t, where m denotes that there are m threads, and ∞ denotes that there is no upper bound on the number of execution resources:

$$1 \leq \pi(m, \infty, t) \leq m, \quad \forall t \in [0, T] \tag{3.1}$$

We define the **inherent program parallelism** $\pi(m, \infty)$, as the number of active threads used by the program, averaged over the entire execution of the program:

Definition 1 (Inherent Parallelism of Shared-Memory Programs). *Given an infinite number of execution resources, the inherent parallelism of a parallel program partitioned into m threads, over its execution time T is:*

$$\pi(m, \infty) = \frac{1}{T} \int_0^T \pi(m, \infty, t) dt \tag{3.2}$$

where $\pi(m, \infty, t)$ is the number of threads performing work at time moment t.

At logical level, the inherent parallelism of a shared-memory program is a measure of the number of threads that can execute concurrently, given no constraints on the number of processor cores available, averaged over the execution time.

At a physical level, when a program consisting of m threads is executed on a set of n homogeneous processor cores, there are two effects on the parallelism of the program:

1. The constraint of the inherent parallelism, if $m > n$, because there are not enough processor cores to execute all the threads concurrently.

2. The presence of runtime overhead which reduces the amount of parallelism that is exploited by the machine, because only part of the work performed by the threads is useful work.

Next we model these two effects.

When a program is partitioned into m threads, only a subset are active at any given time, due to data dependencies. Let $\pi(m, n, t)$ denote the number of active threads running on n cores at time t, and $\pi(m, n)$ denote the number of active threads, averaged over the entire execution time of the program, T. Thus,

$$\pi(m, n) = \frac{1}{T} \int_0^T \pi(m, n, t) dt \tag{3.3}$$

When there are enough cores to execute all threads concurrently, $m \leq n$, the average number of active threads is maximized, and is equal to the inherent parallelism $\pi(m, \infty)$. Thus, $\pi(m, \infty) = \pi(m, n \geq m)$. However, if the number of cores $m > n$, then the number of active threads is constrained by n.

We define the exploited parallelism to account for the effect of bounding the number of existing cores and runtime overhead:

Definition 2 (Exploited Parallelism of Shared-Memory Programs). *Given a finite number of processor cores n, the exploited parallelism of a parallel program partitioned into m threads, $\pi'(m,n)$, with execution time T, is:*

$$\pi'(m,n) = \frac{1}{T}\int_0^T \pi'(m,n,t)dt \tag{3.4}$$

where $\pi'(m,n,t)$ is the number of cores performing useful work at time t.

When the number of cores n on which the program is running increases towards m the exploited parallelism becomes less constrained by the number of available cores. Finally, when $m \leq n$, there is no longer any constraint of the parallelism, because m cores are enough to execute all threads concurrently. We mark this as $\pi'(m,\infty) = \pi'(m, n \geq m)$. However, $\pi'(m,\infty)$ is different from $\pi(m,\infty)$ due to runtime overheads.

The inherent and exploited parallelism are interesting to both users and developers of parallel programs. The inherent parallelism upper bounds the speedup obtainable using any number of cores, in the absence of superlinear speedup effects [34]. The exploited parallelism represents the amount of useful work extracted from a program by a parallel machine. When executed on a real machine, the program execution consists of both useful work and runtime overheads. This thesis focuses the runtime overhead study due to memory contention. Although there are many types of runtime overheads in parallel systems [21], a significant type of runtime overhead in multicore systems is caused by contention among cores for shared resources [63, 74, 75, 98].

3.2.2 Response Times and Parallelism

Next we define the useful work of a program, and derive the equation of exploited parallelism as a factor of useful work of the program, data-dependency, memory contention and I/O overheads.

The key idea behind the approach is to express the exploited parallelism and the parallelism loss such that we can relate it to the causes of parallelism loss. We have used the CPU activity as the handle for expressing the useful work of a program. We divide the activity of a CPU core into four parts, shown in

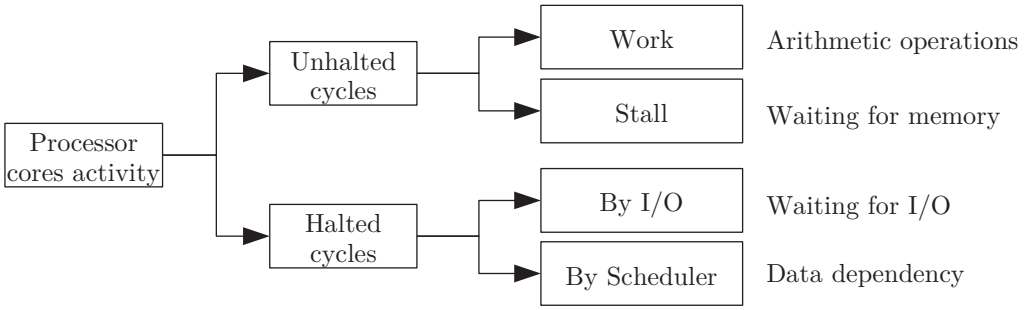


Figure 3.5: CPU core activity and overheads

figure 3.5, and explain below how each part relates to the useful work and the parallel overheads of a program. At any time moment, a CPU core can be either halted or unhalted. An unhalted core is executing the code of the program. In this case, the core can execute work cycles, during which at least one integer or floating point instructions is retired, or it can issue a memory request. The core can also be waiting for a memory request to finish, and if it does not have any arithmetic instructions to execute at the same time, it will incur a stall cycle. A halted core is prevented by the OS scheduler to execute. We distinguish two reasons for a halted core. First, the OS is waiting for an I/O operation issued by this core to finish, and the core does not have any instructions to overlap with this waiting time. Second, the OS scheduler has stopped the thread because of an

operation on a mutex, semaphore, conditional variable or barrier. We use these categories of activities to define our useful work and the three types of parallel overheads.

In our model, the applications consist of workloads that are serviced by three types of resources: n CPU cores, the memory subsystem and one I/O device. However, the response times required at these resources can overlap in time, and thus, the response time of the program cannot be established by simply adding the service and waiting time on all the resources. Modern server systems, including ones based on low-power processors, have I/O devices that can send and receive data without intervention from the CPU cores. They do so because the device is memory mapped, and all data transferred between the device is marshalled by a special processor called a DMA controller. Therefore, I/O device response time can be overlapped with the CPU response time. Furthermore, CPU cores have deep pipelines supporting out-of-order executions that overlap the execution of arithmetic operations with the waiting for memory requests. Thus, in a multicore system, the arithmetic instructions executed by multiple cores in parallel are overlapped with waiting for outstanding memory requests and I/O response time on the I/O device. However, from a measurement point of view, not all active and idle times are independent. A CPU core is seen as active both during execution of arithmetic operations, and while waiting for memory requests [49]. As a result, the *CPU time* of a program effectively accounts for both memory response time and service time of arithmetic instructions.

Based on the overlap between processor and I/O resources, we define two response times in the system:

1. *CPU response time* (C) – total time during which a core is executing instructions or waiting for memory requests, for all cores;

2. *I/O response time* (I) – total time during which a core is waiting for the I/O device, for all cores;

First we build the model for the CPU response time. Later we extend this model to include the I/O response time.

CPU Response Time

Let W be the service time of the CPU activity of the program, expressed as time units, and let M be the total response time of all the memory requests performed by the program. Considering the overlap between the arithmetic operations and the waiting for memory requests:

$$C(n) = \begin{cases} W(n) & \text{if the cores are the bottleneck;} \\ M(n) & \text{if the memory is the bottleneck.} \end{cases} \tag{3.5}$$

which is equivalent to

$$C(n) = max\Big(W(n), M(n)\Big)$$

Because we consider weak-scaling programs, the useful work of a program W does not change with n. In modeling the memory contention (section 3.2.5), we provide more details on this hypothesis, and in the validation section we provide experimental evidence to support it.

However, the response time of the memory requests can increase when the number of active cores n increases. As such, we can express M:

$$M(n) = M(1) + \Delta M(n) \tag{3.6}$$

where $M(1)$ is the response time of the memory requests when only one core is active, and $\Delta M(n)$ is the delay caused by the memory contention among the n

cores. Because $\Delta M(n)$ is a measure of memory contention among cores, when activating multiple cores, we consider that CPU response time is monotonically increasing as long as the memory configuration does not change. Thus, $\Delta M(n) \geq 0$. When the cores are the bottleneck, there will be no increase in the CPU time, and $\Delta M(n) = 0$. But when the memory is the bottleneck, $\Delta M(n) > 0$.

Without losing generality, we can express the two cases of $C(n)$ as:

$$C(n) = max\Big(W, M(1) + \Delta M(n)\Big) \tag{3.7}$$

On an execution using one core, $C(1) = max\Big(W, M(1)\Big)$. Thus, we can rewrite equation 3.7 as:

$$C(n) = C(1) + \Delta M(n) \tag{3.8}$$

where $C(1)$ is the service time of the useful work of the program, and $\Delta M(n)$ is the memory contention overhead.

The total service time of the program is thus:

$$\int_0^{T(n)} \pi(m, n, t)dt = C(n) = C(1) + \Delta M(n) \tag{3.9}$$

Equation 3.8 shows that the service time of a parallel program depends on the overlap between useful work and waiting for memory requests: *When the cores are the system bottleneck, the service time of the entire program is equal to the service time of the useful work. In contrast, when the memory is the system bottleneck, the service time of the program is equal to the response time of the memory requests.*

I/O Response Time

Next we extend the model for programs with network I/O requests. Because we consider that the I/O device is memory-mapped and consider that the DMA

controller marshals the data between the I/O device and the memory, we assume the CPU is not utilized during any I/O transfer. This allows the I/O transfers to be overlapped with both useful work and with memory contention.

If $I(n)$ is the total I/O waiting time seen by the cores, then the total service time of the program consists of the sum of all the moments when the program is executing useful work, executing stall cycles due to memory contention, and waiting for I/O transfers to finish. Because the I/O transfer can overlap with the useful work and the waiting for memory accesses, we note that *only I/O transfers that are not overlapped to useful work are considered I/O overhead.* Thus, the total service time of a program with I/O overhead is:

$$\int_0^{T(n)} \pi(m,n,t)dt = max\Big(I(n), C(n)\Big) = max\Big(I(n), C(1) + \Delta M(n)\Big) \tag{3.10}$$

Let $\Delta I(n)$ be the response time of the I/O that is not overlapped with the useful work $T(1)$:

$$\Delta I(n) = I(n) - T(1) \tag{3.11}$$

The service time of a program can be rewritten as:

$$\int_0^{T(n)} \pi(m,n,t)dt = T(1) + \Delta M(n) + \Delta I(n) \tag{3.12}$$

3.2.3 Useful Work

Taking a pragmatic view on parallel overheads, we note that any execution on one core does not suffer from any parallel overhead. As such, we can define the useful work of a parallel program, U as the response time of the program incurred during

an execution on one core:

$$U = T(1) \tag{3.13}$$

Due to the overlap between the service times of the three resources:

$$U = max\Big(W, M(1), I(1)\Big) \tag{3.14}$$

Next we derive the execution time of the program on an arbitrary number of cores, $T(n)$ and the parallel speedup.

From the definition of $\pi(m, n)$, it follows that the execution time of the program on n cores, $T(n)$ is:

$$T(n) = \frac{\int_0^{T(n)} \pi(m, n, t)dt}{\pi(m, n)} = \frac{T(1) + \Delta I(n) + \Delta M(n)}{\pi(m, n)} \tag{3.15}$$

On a single core, there is no extra delay caused by memory contention among cores, hence $\Delta M(1) = 0$. Additionally, on a single core, $\pi(m, 1) = 1$, because there can be a single active core. The total amount of time required to execute the program on one core, $T(1)$, is:

$$T(1) = U$$

The parallel speedup is therefore:

$$\frac{T(1)}{T(n)} = \pi(m, n)\frac{U}{U + \Delta M(n) + \Delta I(n)}$$

We normalize the waiting time for memory and I/O to the useful work time, $U = T(1)$. We define the memory contention factor, ω_μ

$$\omega_\mu(n) = \frac{\Delta M(n)}{U} \tag{3.16}$$

the ratio of average number of active threads due to memory contention to the average number of active threads due to useful work. We further define the I/O overhead factor, ω_σ

$$\omega_\sigma(n) = \frac{\Delta I(n)}{U} \tag{3.17}$$

as the ratio of average number of threads waiting for I/O to the average number of threads active due to useful work.

Thus, the speedup of a shared-memory program with m threads on n cores is:

$$\frac{T(1)}{T(n)} = \frac{\pi(m,n)}{1+\omega_\mu(n)+\omega_\sigma(n)} \tag{3.18}$$

We show next that the parallel speedup is equivalent with the exploited parallelism. At any time moment t, we have $\pi'(m,n,t)$ threads executing useful work, $\omega_\mu(n)\cdot\pi'(m,n,t)$ threads active due to memory contention and $\omega_\sigma(n)\cdot\pi'(m,n,t)$ active due to I/O overhead. Therefore, the average number of threads performing work, $\pi(m,n)$, observed over the execution time $T(n)$ is:

$$\pi(m,n) = \frac{1}{T(n)} \int_0^{T(n)} \Big(1+\omega_\mu(n)+\omega_\sigma(n)\Big)\pi'(m,n,t)dt$$

and we have

$$\pi'(m,n) = \frac{\pi(m,n)}{1+\omega_\mu(n)+\omega_\sigma(n)} \tag{3.19}$$

Equations 3.18 and 3.19 show the conceptual equivalence of our definition of exploited parallelism to the parallel speedup: $\int_0^{T(n)} \pi'(m,n,t)dt$ is equivalent to execution time on one core, without parallel overheads, and $T(n)$ is the execution time on n core, including the parallel overheads. However, expressing the parallelism performance as inherent and exploited parallelism allows us to quantify the factors that affect performance as parallelism loss.

Equation 3.19 exposes a useful insight on the parallelism of a shared-memory program. For a program partitioned into m threads and executed on n cores, $\pi(m, n)$ is the total amount of parallel work, and the denominator expresses how much of that work is useful work.

Eager et. al [34] have derived a expression similar to equation 3.19, using the average program parallelism and the speedup, but without considering runtime overhead. In general, our speedup model is consistent with many studies on inherent parallelism [34, 101, 54, 32]. However, these studies relate the speedup only to the average parallelism of the program and express the performance loss due to constraining the number of cores. These simplified models, while useful, are impractical for understanding current multicore systems, where memory contention can have a significant impact on speedup performance. Another limitation of existing studies based on inherent parallelism is that they do not provide a method of deriving the inherent parallelism. Simply measuring the active number of threads is not sufficient, since some of the threads may be active but stalled due to memory contention. In contrast, our model defines the speedup loss due to data dependency and memory contention. Furthermore, we provide a practical approach of deriving $\pi(m, n)$, $\omega_\mu(n)$ and $\omega_\sigma(n)$.

3.2.4 Data Dependency

Next we present a model to determine $\pi(m, \infty)$, $\pi(m, n)$ and the parallelism loss due to data dependency, $\pi_d(m)$.

The data dependency of the program is the average number of threads which are inactive throughout the execution time of the program, if there are no constraints on the number of execution resources. The reasons why some threads may not be active include synchronization operations, load imbalances among threads and insufficient work to keep all threads busy. In general, we treat all sources

of thread inactivity as data dependency. We acknowledge that there are reasons unrelated to the parallelism that may cause threads to become inactive, such as I/O operations or some system calls, even in the absence on constraints on the execution resources. As we target applications with heavy memory contention and data dependency, we focus on parallel programs where system calls do not represent a significant source of performance loss.

Definition 3 (Parallelism Loss Due to Data Dependency). *The parallelism loss due to data dependency in the program, $\pi_d(m)$, is defined as the difference between the total number of available threads, m, and average number of active threads, $\pi(m, \infty)$, given no constraints on the number of execution resources*:

$$\pi_d(m) = m - \pi(m, \infty) \tag{3.20}$$

We derive a model for the data dependency of the program that determines the average number of active threads of a program, $\pi(m, n)$, based on an execution of the program partitioned into m threads on b cores, where $m > b$. Based on $\pi(m, n)$, we then determine $\pi_d(m)$.

The insight behind our approach is that when the number of threads is greater than the number of cores, the sum of the number of executing threads and the number of threads in the run-queue represents the number of active threads of the program. Our key idea is to conduct one measurement run, called a *baseline run* executing the program on a number of processor cores smaller than the number of threads. Since there are more threads than cores, some of the threads will queue for service in the run-queue. Based on the profile of number of active threads over time and on the service time of the threads, we infer the time required to execute the threads when there are enough cores to execute all the threads concurrently,

$m \leq n$, without considering the memory overhead. We then determine $\pi(m, n)$ as time weighted average.

However, oversubscribing the cores has three effects on performance [55]:

1. The total execution time of the program is different compared to when there are enough cores for all the threads, because there are not enough cores to execute all threads concurrently.

2. It may cause load imbalances between threads.

3. It may cause significant context switching which may increase the kernel service time of the program as well as affect the efficiency of the caching.

Our model accounts for the first two effects, and we give an experimental analysis of the third.

We run the program partitioned into m threads on b cores, where $m > b$. We determine the time required to finish the program, given enough cores to execute all threads concurrently, $m \leq n$, which we denote as critical path time T_{cp}. The average number of active threads is then determined as a time weighted average of the parallelism of each region of program.

When $m > n$, some threads may have to queue for service in the run-queue. Therefore, at time moment t, $x(m, t)$ denotes the number of threads that are executing and $q(m, t)$ denotes the number of threads that are queueing. Given enough cores to run all the threads in parallel, $m \leq n$, then the number of active threads is:

$$\pi(m, \infty, t) = x(m, t) + q(m, t)$$

Based on the time required to execute the baseline run, the model determines the critical path time, T_{cp}.

During the baseline run, let τ denote the service time received by the program

from time t to time $t+\Delta T$. If ΔT is sufficiently small such that there is no change in the number of active threads from t to $t+\Delta T$, we have:

$$\tau = x(m,t) \cdot \Delta T = \sum_{j=1}^{m} \tau_j$$

where τ_j is the service time required by thread j. The execution time when $m \leq n$, is ΔT_{cp}, and is equivalent to the maximum service time received by one thread:

$$\Delta T_{cp} = max\{\tau_j\}$$

The average number of active threads during this interval is:

$$\pi(m,\infty,t) = \frac{\tau}{\Delta T_{cp}} = \frac{\sum_{j=1}^{m} \tau_j}{max\{\tau_j\}}$$

When $m \leq n$ cores, the total execution time of the program, T_{cp}, is the sum of the minimum time to execute every part of the program $T_{cp} = \sum \Delta T_{cp}$. Therefore, the average number of active threads over the entire execution of the program is the average of the values of $\pi(m,\infty,t)$ with weights ΔT_{cp}:

$$\pi(m,\infty) = \frac{\sum \pi(m,\infty,t)\Delta T_{cp}}{T_{cp}} \tag{3.21}$$

and

$$\pi_d(m) = m - \frac{\sum \pi(m,\infty,t)\Delta T_{cp}}{T_{cp}} \tag{3.22}$$

It can be argued that since $\tau = x(m,t) \cdot \Delta T$, it is not needed to measure τ_j. However, simply computing ΔT_{cp} as an average, $\Delta T_{cp} = \frac{x(m,t)\cdot\Delta T}{x(m,t)+q(m,t)}$, would not account for the load imbalances between threads. This may lead to underestimating the data dependency of the program.

Next, we show the derivation of the average number of active threads when

the program is executing m threads on an arbitrary n cores. We start from the profile of the active number of threads $\pi(m, \infty, t)$ over time. If during interval ΔT there are $\pi(m, \infty, t)$ active threads, then on n cores the number of executing threads is $min\{n, \pi(m, \infty, t)\}$. Therefore, the time required to execute them on n cores, $\Delta T(n)$, is:

$$\Delta T(n) = \frac{\Delta T_{cp} \cdot \pi(m, \infty, t)}{min\{n, \pi(m, \infty, t)\}}$$

The average number of active threads, $\pi(m, n)$ is determined as the average number of active threads, $min\{n, \pi(m, \infty, t)\}$, weighted to $\Delta T(n)$ for part of the program, similarly to equation 3.21:

$$\pi(m, n) = \frac{\sum \pi(m, n, t)\Delta T(n)}{\sum \Delta T(n)} \tag{3.23}$$

We implemented the model using sampling. The size of the run-queue is sampled using a constant ΔT time interval. For every sample, we measure $\pi(m, \infty, t)$ and the vector of service time of the threads, τ_j.

The proposed model for determining the average number of active threads and the data dependency of the program has important practical advantages. The proposed approach is independent of programming languages, threading package or programming methodology because we use the dynamic size of the run-queue as the proxy for determining the parallelism. Furthermore, the OS statistics about the dynamic size of the run-queue are be obtained using a non-invasive method, such as reading the `procfs` pseudo-file-system. This entails that we do not need any type of instrumentation of program source or binary code, which confers important practical properties. Program instrumentation is intrusive and often prevents some types of optimizations. For example, OPARI which is at the heart of KOJAK [119], TAU [102], Scalasca [46] and ompP [39] performance analysis tools, prevents the usage of implicit barriers, which in turn prevents the OpenMP

NOWAIT clause, thus forcing the threads to perform an additional synchronization operation. Instrumentation may slow down the program or interfere with cache sensitive areas, and therefore increase the overhead of the parallel programs. Some vendors provide highly optimized versions of popular parallel kernels (such as BLAS or LAPACK) which come directly compiled as libraries. Without access to the source code of such products, prediction methods relying on instrumentation may be inapplicable. Lastly, instrumentation methods lack generality, because often they cannot be applied across different programming languages, threading packages and runtime systems.

3.2.5 Memory Contention

We propose a model to derive the memory contention factor, $\omega_\mu(n)$, and parallelism loss due to memory contention among cores for shared-memory programs. We previously defined the memory contention factor $\omega_\mu(n)$, the ratio of average number of active threads due to memory contention to the average number of active threads due to useful work:

$$\omega_\mu(n) = \frac{\Delta M(n)}{U} \tag{3.24}$$

Definition 4 (Parallelism Loss Due to Memory Contention). *The parallelism loss due to memory contention among cores, π_μ, is defined as the number of threads busy due to memory contention, averaged over the entire execution of the program:*

$$\pi_\mu(m,n) = \pi(m,n)\frac{\Delta M(n)}{T(n)} \tag{3.25}$$

From the definition of memory contention factor, ω_μ, we can rewrite the par-

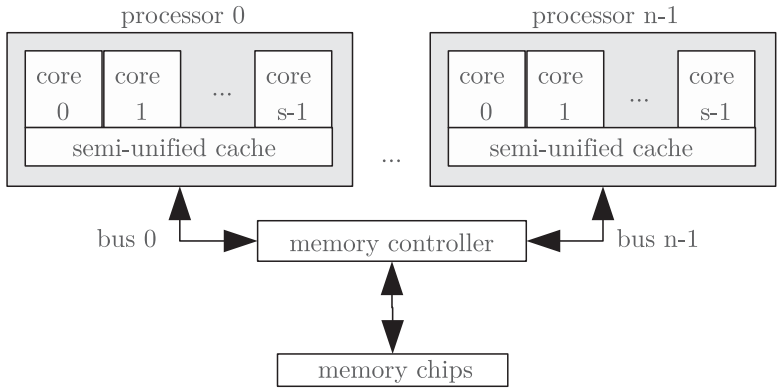
allelism loss due to memory contention, $\pi_{mu}(m, n)$ as:

$$\pi_{\mu}(m, n) = \pi(m, n)\frac{\omega_{\mu}(n)}{\omega_{\mu}(n) + \omega_{\sigma}(n) + 1}$$

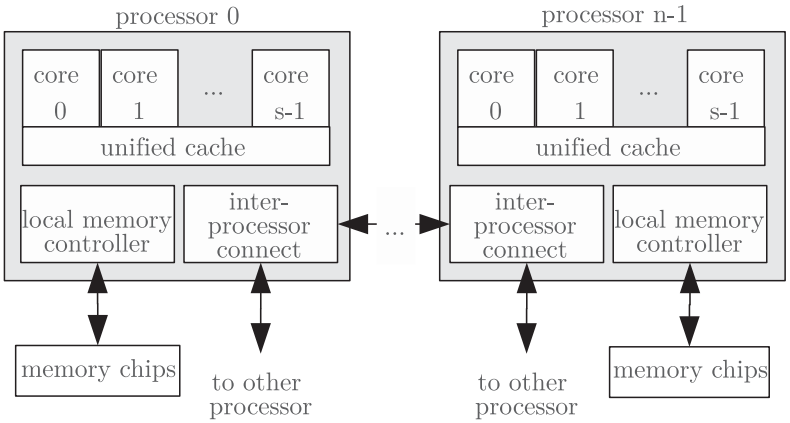
Next we derive a general model for memory contention factor.

The targeted architectures are considered multiprocessors of multicores. We model the general cores as having variable frequency with out-of-order superscalar or pipelined execution, and contain multiple level of caches which can be private (per-core) or shared among cores. The caches are considered inclusive, therefore only the last-level cache misses are sent to the memory controllers to be serviced. The interconnect architectures between processor are uniform memory access (UMA) and non-uniform memory access (NUMA). Figure 3.6 shows the characteristics of the UMA and NUMA memory architectures. In UMA, each processor has a dedicated bus to the single memory controller, while in NUMA each processor has its own bus to the local memory controller, as well as a connection to any other processor. This simple model covers previous generation and state-of-the-art multicore systems, based on Intel Core microarchitecture and ARM Cortex microarchitectures (UMA), and Intel Nehalem and AMD K10 (NUMA). For UMA, we consider the number of processors as maximum two, while for NUMA we do not bound the maximum number of processors.

We propose a model for the memory contention factor based on quantifying the number of cycles spent by a program on a multicore systems. Our model is centered on the memory contention among different cores. Unlike many existing studies [62, 120], we are not interested in the absolute value of processor cycles, but in the growth of the processor cycles due to memory contention among cores. We therefore are interested in the growth of number of cycles relative to a baseline value on one core, where there is no memory contention among cores.



(a) n processors with UMA interconnect



(b) n processors with NUMA interconnect

Figure 3.6: Architectures of multi-processor multicore systems: UMA & NUMA

Without losing generality, we simplify the derivation of the memory contention model by focusing on the cases where the cores or the memory are the system bottleneck, and the program has no I/O requests. Thus $\omega_\sigma = 0$ and $U = T(1) = C(1)$. Because we consider out-of-order cores, we model the overlap between executing useful work and waiting for memory requests. Previously we defined the service time of an execution phase, C, as the overlap between useful work and waiting for off-chip requests:

$$C(n) = max\Big(W(n), M(n)\Big)$$

We divide the cycles into three categories:

1. Work cycles: $w(n)$, which are cycles in which at least one floating point or integer operation is retired, or at least one memory operation is issued;

2. Stall cycles that are not due to resource contention, such as pipeline hazards, branch mispredictions, cache hits and uncontended memory accesses: $b(n)$;

3. Cycles spent waiting for off-chip memory accesses: $a(n)$.

If the core frequency during an execution episode is f, let $c(n)$ denote the total number of cycles incurred by n cores when executing a shared-memory program. During the execution of an episode, we have:

$$c(n) = f \cdot C(n) \tag{3.26}$$

$$w(n) + b(n) = f \cdot U(n) \tag{3.27}$$

$$a(n) = f \cdot M(n) \tag{3.28}$$

With these notations, we can rewrite $\Delta M(n)$ as:

$$\Delta M(n) = \frac{\Delta C(n)}{f} = \frac{a(n) - a(1)}{f} = \frac{c(n) - c(1)}{f} \tag{3.29}$$

Due to the overlap between executing useful work and waiting for memory requests, we have:

$$c(n) = max\Big(w(n) + b(n), a(n)\Big)$$

Because we target weak-scaling workloads, from measurement analysis we observe that $w(n)$ and $b(n)$ are constant with n, when the number of cache misses and the total number of instructions do not change with n. While it is expected that w does not depend on n, we comment here on why b does not change when n changes. The intuitive explanation for this behavior is that, since $b(n)$ represents

the stalls due to uncontended resources, it does not matter how many cores split these stalls, because their total remains the same. Similarly, the execution time of floating point and integer instructions depends only on the availability of the operands. If caching does not change when n changes, then operand availability does not change, and neither does the number of cycles required to execute them. Moreover, we are interested in modeling the number of total cycles for large program runs, with long steady-state compute phases that result in balance of shared caches. Therefore small, transient deviations from this assumption do not make the objective of this study. Furthermore, we have evaluated this assumption empirically on all our workload tests and present evidence to support it.

On a single core, the contention for shared resource among cores is zero, therefore $c(1) = max\big(b(1) + w(1), a(1)\big)$ can be considered the useful work part of the program. Although $b(1)$ are stall cycles, we consider them a component of the useful work, i.e. the number of stall cycles required to fetch the data when there is no contention among cores. Considering that the frequency f is fixed during the execution of one program episode:

$$\omega(n) = \frac{\Delta M(n)}{U} = \frac{f\big(M(n) - M(1)\big)}{w + b} = \frac{a(n) - a(1)}{w + b} = \frac{c(n)}{c(1)} - 1 \tag{3.30}$$

The total number of cycles, $c(n)$ is modeled using a hierarchical approach, as follows. We derive first an equation for $c(n)$ for one socket and subsequently we model the effect of interconnecting multiple sockets.

The cores have inclusive caches, therefore only last level misses are sent to the memory controllers. Furthermore, because we are interested in large program sizes, we consider that the overwhelming type of cache misses are data-cache misses. Once a memory request misses the last-level data cache, it is sent to the memory

controller to be serviced and then the data returns to the last-level cache. There are several queues in which the memory request might be kept from the moment it leaves the last-level cache to the moment it returns to the last-level cache, and the queueing discipline in each of the memory bus queues or memory controller queue is difficult to measure or analyze directly. There are very few information in the literature about the queueing disciplines of various types of buses and memory controllers for Intel, AMD or ARM systems. Furthermore, the differences between the targeted architectures are significant, and even if all the queueing time could be modeled directly for each of the queue, and thus derive a general equation for the queueing time, solving this equation to a closed-formed might be impossible [59]. We therefore favor a simplifying assumption that allows a unified approach to the memory systems of all the architectures, and solving the equation to a closed-form solution.

For both UMA and NUMA, we consider one memory controller to be a single server system with one queue, in which requests are serviced in first-come-first-serve order. We apply a single-server queueing model to determine the response time of the memory requests, $M(n)$. From our experiments, which are detailed in section 4, we identify two cases of memory requests patterns:

1. If the memory is the system bottleneck, the memory requests are non-bursty, and an exponential distribution fits their arrival rate over time. In this case, the memory subsystem is treated as an M/M/1 queueing model.

2. If the memory is not the system bottleneck, the memory requests are bursty and a Pareto distribution fits their arrival rate over time. We use an M/D/1 queueing system to model the response time of the memory requests.

Out of these two cases, the first one is more valuable, because the CPU response time is be dominated by the response time of the memory requests. For the second

case, because the bottleneck is not the memory, the response time for the entire program will not be dictated by either the I/O response time or the response time of the useful work.

Memory is System Bottleneck

We apply a $M/M/1$ model to derive the response time of the memory requests. Since the memory requests start from the cores but are filtered by two or three levels of cache, the inter-arrival times of the request are assumed independent and identically distributed. Considering the superscalar and out-of-order nature of modern processors, cores issue memory requests and flops/integer operations instructions at the same time. This means that while cores are waiting for memory requests to be completed, they are also executing instructions for which the operands have been fetched from memory. Therefore, for programs with significant memory contention, *the critical path of the execution time of a program is dominated by the response time of memory requests.* Let $M_{req}(n)$ be the average response time of one memory request that has arrived at a memory controller which services n cores. From the $M/M/1$ model [59]:

$$M_{req}(n) = \frac{1}{\mu - \lambda}$$

where μ is the service rate of the memory controller and λ is the arrival rate of the memory requests. Let $r_M(n)$ denote the total number of last level cache misses, and L the arrival rate of requests from one core. Because the response time of the memory node is independent of the core frequency f, the total number of CPU cycles incurred while waiting for one memory request is:

$$a_{req}(n) = f \cdot M_{req}(n) = \frac{f}{\mu - \lambda}$$

For a single socket system, with n cores active, $\lambda = n \cdot L$ and the total number of cycles incurred by the program is

$$a(n) = r_M(n) a_{req}(n) = \frac{f \cdot r_M(n)}{\mu - n \cdot L} \tag{3.31}$$

and the total time spent by a the processor cores waiting for memory is:

$$M(n) = \frac{r_M(n)}{\mu - n \cdot L} \tag{3.32}$$

Because the system bottleneck is the memory:

$$c(n) = a(n) = \frac{f \cdot r_M(n)}{\mu - n \cdot L} \tag{3.33}$$

Next we extend the model to multiple processors, using a *fill-socket-first* policy. Let n_s be the number of cores of one processor. In UMA, each processor has its own bus, and therefore requests from different sockets queue for memory access separately. Therefore, queueing time is modeled separate for each processor. In a two socket system, if n_1 cores are active in the first socket and n_2 in the second, $a_{UMA}(n) = a(n_1) + a(n_2) + \Delta a$, where Δa represents the increase in number of cycles due to the increase in load on the controller, which services requests from two sockets, instead of one. If n changes using a fill socket first policy, when changing from n_s cores (all on the first socket) to $n_s + 1$ cores (n_s on the first processor and one on the second processor), the difference between $\Delta a = a(n_s+1) - a(n_s)$ reflects the increase in response time from increasing the load on the memory controller:

$$a_{UMA}(n) = c(n_s) + c(n_s - s) + \Delta c \tag{3.34}$$

For NUMA, when two processors and two memory nodes are active, there

is an additional delay to send the memory request to a remote node. Let δ be the additional time required to send one memory requests to a remote memory controller, compared to the case when only the local controller is active. A core splits its memory request into local memory request and remote memory requests. The total number of cycles required for a memory request to be serviced by one memory controller that services n cores is:

$$\begin{aligned} a_{local}(n) &= a(n) \\ a_{remote}(n) &= \delta(n) + a(n) \end{aligned}$$

and δ depends on the ratio of remote memory accesses to total memory accesses.

We use a linear model for δ. If n cores active are split as n_s on the local socket and $n - n_s$ cores on the remote, on average the memory accesses will be split $\frac{n_s}{n} r_M(n)$ on the local memory controller and $\frac{n-n_s}{n} r_M(n)$ on the remote memory controller. Therefore, the total number of cycles for a two-processor NUMA system is:

$$a_{NUMA}(n) = \frac{n_s}{n} r_M(n) a_{local} + \frac{n - n_s}{n} r_M(n) a_{remote} \tag{3.35}$$

In equations 3.31, parameters L and μ implicitly model the effect of memory request and memory performance on the number of cycles. Similarly, in equations 3.34 and 3.35, the parameters Δa and δ account for the increase in cycles when activating additional sockets. A detailed model of these parameters is beyond the objective of this thesis, because a memory bus and memory controller performance model is platform specific. Furthermore, from a practical point of view, parameters L and μ can be extracted using linear regression from a set of measured values of $a(n)$. Since we have observed that $r_M(n)$ is constant, parameters L and μ can be linearly regressed using equation 3.31 and at least two points of $\frac{1}{a(n)}$. Similarly, Δc and δ can be determined using measured values of $a(s)$ and

$a(s+1)$. Because the memory is the system bottleneck, $c(n) = a(n)$, and thus we can replace all measurements of the cycles incurred while waiting for memory, a, with measurements of the total number of cycles incurred by a program, c. Therefore, to apply the memory model, we need the following set of input parameters:

1. For a single socket system, two runs of the program on n_1 and n_2 cores and measurements of $c(n_1)$ and $c(n_2)$ are required. Parameters L and μ are regressed through the coordinates $\{n_1, 1/c(n_1)\}$ and $\{n_2, 1/c(n_2)\}$.

2. For a multiple socket system, is required:

   - On UMA, measurements on 1, n_s and $n_s + 1$ cores. We measure $\Delta c = c(n_s + 1) - c(n_s)$, in addition to regression of L and μ.
   - On NUMA, measurements on 1, n_s and $n_s + 1$ cores. Parameter δ is regressed from the line $\{n_s, c(n_s)\}$ to $\{n_s + 1, c(n_s + 1)\}$, in addition to L and μ.

Therefore, $\omega(n)$ can be determined from at most three measurements of $c(n)$ via equation 3.30.

Cores are the System Bottleneck

The memory requests that are send to the memory server are filtered by two levels of cache, and since the programs analyzed in this paper consist of server workloads, we assume that there is sufficient time between memory requests arrivals to satisfy a memorylessness property of the memory requests inter-arrival time. However, we do not make any assumptions on the size of the memory requests, and thus, on the distribution of service times required by the memory requests. $T_{M,j}$ is composed of service time $S_{M,j}$ and waiting time $Z_{M,j}$ of the memory requests.

$$T_{M,j} = S_{M,j} + Z_{M,j} \tag{3.36}$$

Let r_M be the total number of last level cache misses, $\bar{r_{M,j}}$ the average number of last level cache misses requested during one instruction window, and $Var(r_{M,j})$ the variance of last level cache misses requested during one instruction window. The total number of instruction windows throughout the execution of the program is j:

$$j = \frac{r_M}{\bar{r_{M,j}}} \tag{3.37}$$

When s_M is the service time required by one memory request, then the average and variance of the service time required by all r_M requests are:

$$\begin{aligned} \bar{S_{M,j}} &= s_M \cdot \bar{r_M} \\ Var(S_{M,j}) &= s_M^2 \cdot Var(r_M) \end{aligned}$$

Let λ_M be the arrival rate of memory request from a single core. If there are n active cores that are issuing memory request, the total arrival rate of memory requests is:

$$\lambda = n \cdot \lambda_M \tag{3.38}$$

The response time of the r_M memory requests is modeled using a M/G/1 queueing system. From Pollaczek-Khinchin formula [110]:

$$S_{M,j}(n) = \bar{r_{M,j}} s_M \tag{3.39}$$

$$Z_{M,j}(n) = \bar{S_{M,j}}^2 \lambda \frac{1 + Var(S_{M,j})}{2(1 - \bar{S_{M,j}}\lambda)} \tag{3.40}$$

From equations 3.37 to 3.40:

$$T_{M,j}(n) = \bar{r_M} s_M + \bar{r_{M,j}}^2 s_M^2 n \lambda_M \frac{1 + s_M^2 Var(r_{M,j})}{2(1 - \bar{r_{M,j}} s_M n \lambda_M)}$$

$$T_M = j \cdot T_{M,j} = r_M s_M \Big(1 + s_M \bar{r_{M,j}} n \lambda_M \frac{1 + s_M^2 Var(r_{M,j})}{2(1 - s_M \bar{r_{M,j}} n \lambda_M)}\Big) \tag{3.41}$$

Equation 3.41 shows that the response time of the memory requests degrades with an increase in n. Furthermore, the increase in memory response time also depends on the burstiness of memory traffic. Equation 3.41 also describes how the workload interacts with the machine: λ_M and r_M depend on the workload, while s_M is a system parameter, which depends on the bandwidth of the memory system.

To apply the model, we determine the response time of one memory request as the ratio of cache line size and the effective memory bandwidth. The average memory burst size $r_{M,j}$ is determined based on the probability profile of the burst size. In the model parameterization section, we show that server workloads fall under two categories, bursty memory traffic and non-bursty memory traffic. For bursty memory traffic we use a Pareto distribution to model $\bar{r_{M,j}}$ and $Var(r_{M,j})$, based on inputs collected during the baseline runs. For non-bursty memory traffic, we use an exponential distribution of burst size, and reduce the M/D/1 model to an M/M/1 model. Finally, r_M is determined from the trace of the hardware events counters.

3.2.6 I/O Overhead

Next we extend our general model to determine the parallelism loss due to the I/O overhead.

Due to the overlap between performing arithmetic work, waiting the memory requests and waiting for I/O operations, we consider I/O overhead only the threads that are waiting for an I/O operation to finish without executing useful work at the same time.

We previously define the I/O overhead factor, ω_σ as the ratio of threads waiting

for I/O operations to the threads performing useful work:

$$\omega_\sigma(n) = \frac{\Delta I(n)}{U}$$

Definition 5 (Parallelism Loss Due to I/O Overhead). *The parallelism loss due to I/O overhead, π_σ, is defined as the number of threads performing I/O operations that are not overlapped with useful computations, averaged over the entire execution of the program:*

$$\pi_\sigma(n) = \pi(m,n)\frac{\Delta I(n)}{T(n)} \tag{3.42}$$

Next we discuss our approach for determining $\Delta I(n)$. From the definition of $\Delta I(n) = I(n) - T(1)$ we note that the waiting time experienced by the n cores on the I/O device depends on the response time of the I/O device, which we denote I_r. When there are n core out of which $\pi(m,n)$ are active, the I/O waiting time cumulated over the entire cores is:

$$I(n) = \pi(m,n)I_r \tag{3.43}$$

To model I_r, the applications targeted by us involve network I/O requests operating based on a *request-reply* pattern. In general, many types of web-hosting workloads are governed by this pattern [110]. For example, a webserver receives an HTTP request on the I/O interface, forms a reply by performing some computations and then sends the reply back to the sender.

The typical mechanism employed by a program to receive data from a I/O device involves performing a system call on a network socket. To service the requests, a thread performs a system call (on Linux, typically `read` or `recvmsg`) instructing the operating system to read the content of the request from the device.

If the system does not receive any data on the device, the system call blocks the calling thread until the request can be completed. When the I/O device receives the request data, the operating system copies the data to the main memory using direct memory access (DMA), and then unblocks the thread from the system call. After the reply is formed, the thread performs another system call (typically `write` or `sendmsg`) that instructs the operating system to send a reply data to the I/O device. Thus, response time of an I/O operation can be divided into:

1. I/O blocking time (I_B) – total time between the thread blocking on a read system call and the time moment when the data arrives from the sender to the I/O device, for all read system calls.

2. I/O transfer time (I_T) – total time required to transfer the data between the I/O device and the main memory.

In contrast to the read operation, the write requests do not incur blocking time until the data arrives to the destination, because this aspect of the communication protocol is controlled independently by the operating system, according to the underlying transport protocol. Figure 3.7 shows a typical sequence of I/O system

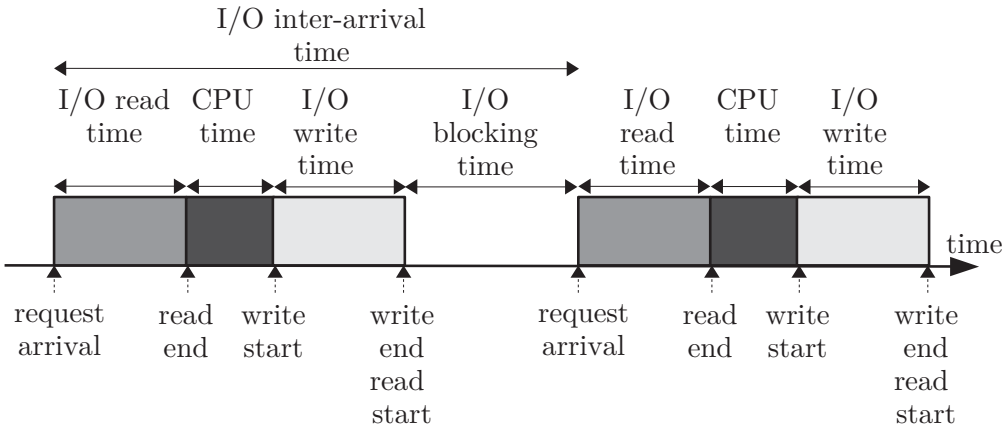


Figure 3.7: Overlapping of I/O times

calls and the I/O blocking times and transfer times. I_T is the sum of the I/O read

and write times. Let $\lambda_{I/O}$ be the inter-arrival time of I/O requests:

$$I_B = \frac{1}{\lambda_{I/O}} - I_T - C \tag{3.44}$$

If we consider $\lambda_{I/O}$ independent of the I/O sequence response time, the blocking time and I/O transfer time of a thread can be overlapped. Thus, for a thread, the response time of the I/O incurred during the request-reply episode is:

$$I_r \approx max(I_T + C, \frac{1}{\lambda_{I/O}}) \tag{3.45}$$

and the I/O idle time, $D_{I/O}$, is the difference between inter-arrival time and the sum of CPU time and I/O transfer time.

However, in server workloads such as *Apache* or *memcached* a thread multiplexes multiple network sockets, such that the CPU time incurred by one request-reply can overlap with the transfer time incurred by another request-reply [110]. Thus, the I/O time of a program is

$$I_r = max(I_T, \frac{1}{\lambda_{I/O}}) \tag{3.46}$$

To apply the I/O model, I_T is determined as the ratio of transferred data to the network bandwidth, and the I/O blocking time is determined from the measured arrival rate of I/O requests. If r_I is the total data transferred by the I/O device and B_I is the I/O device bandwidth, then the transfer time of the I/O device is:

$$I_t = \frac{r_I}{B_I} \tag{3.47}$$

3.3 Power and Energy Models

The objective of the energy model is to predict the power and energy requirements of a program, as a factor of the number of active cores n and core frequency f. We focus the model on the power and energy of the processor cores and of the memory. To derive the energy utilized by a system we use a hybrid measurement-modeling approach, in which we combine measure values of power with modeled service time of the cores, memory and I/O devices. The power measurements are performed only once per system, because they do not depend on the workload. Using information derived from these measurements, we model the energy requirements a program running on a multicore system.

Let E be the energy requirement of a program, P the average power consumption of the program, and T the execution time of the program:

$$E = P \cdot T \tag{3.48}$$

We derive P using a model of process cores, memory and I/O devices. Next, we use the parallelism model to derive the execution time for different core frequencies f, thus completing the energy model.

3.3.1 Power Model

We consider all devices in a system, except the processor, to have two power states: *idle* and *under load*. The processor is considered to have an idle power state, which corresponds with no active cores and several loaded states, which depend on the number of active cores. Each core is considered to have the same two power states. By convention, we note that the processor power utilization under idle load includes *all the idle system power*, including cooling devices, peripheral

devices such as video and storage, and all motherboard circuitry different from the memory and I/O subsystems.

The power drawn by the multicore system at any given time is divided into three components:

1. *Processors power*, P_{CPU}, is defined as the average power drawn by the processor cores, including the power required for cooling them when active.

2. *Memory power*, P_M, is the average power drawn by the memory subsystem.

3. *I/O power*, $P_{I/O}$, is defined as the average power consumption of the network I/O device.

$$P = P_{I/O} + P_M + P_{CPU}$$

Similarly to the parallelism model, the focus of the energy model is programs with large compute, memory or network I/O requirements, and therefore, we do not address programs with significant energy variations caused by storage. Thus, we consider all power drawn by storage and other motherboard circuitry constant, and accounted into P_{CPU}.

Using measurement analysis, we have observed that the memory traffic is non-bursty when the memory is the system bottleneck. From this observation, it follows that the memory bandwidth is utilized constantly throughout the execution of the program. Therefore, we model the memory utilization as fully utilized during periods when any cores is issuing memory requests, and zero utilized when no cores are issuing memory requests.

Furthermore, the memory power is not considered to be affected by the frequency of the processors. For NUMA systems, let p denote number of active

memory nodes and P_{mc} denote the power drawn by one memory node:

$$P_M = p \cdot P_{mc}$$

For UMA, $p = 1$.

On a processor system with n cores, we consider that the cores are utilized only when a subset of the m threads of the programs are active. Therefore, the average core utilization, cumulated across all cores is $\pi'(m,n) + \pi_\mu(n)$. Let P_c be the average power utilization of one core. The power consumption of n active cores is:

$$P_{CPU} = \pi(m,n)P_c$$

Summarizing, the power consumption of a program partitioned into m threads running on n cores is:

$$P(n) = P_i + pP_{mc} + \pi'(m,n)P_c \tag{3.49}$$

P_c includes the power consumption of the core execution resources, such as pipelines, branch predictors, ALU units, floating point units, among others. P_{mc} include the power consumption of the memory controllers, the memory bus, inter-processors communication network and memory banks. Depending on the workload, only some of these components are active. As such, the power drawn by a core depends on the type of workload (i.e. integer operation, floating point operations or memory operations) and the core clock frequency f. We do not model these further, and instead we perform measurements during which we measure the power drawn when a core is effecting floating points, integer and stall cycles, for all supported core frequencies. Similarly, we do not model further the power drawn by the memory in idle and loaded states, but we measure it. Finally, for

I/O, we measure the power under idle and loaded states. These measurements are discussed in the Section 4.3.3.

3.3.2 Energy Model

The energy model predicts the energy used (E) used as a factor of number of cores, n and core clock frequency f. The approach is to divide the energy used based on the three types of resources: cores, memory and I/O device. The total energy of the system, E is:

$$E = E_{CPU} + E_M + E_{I/O} \tag{3.50}$$

The energy used by the cores depends on how the number of active cores and the type of activity effected by the cores. Let $P_{CPU}(n, f)$ be the power drawn by the processor when n core are active and operating at frequency f. By convention we denote $P_{CPU,idle} = P_{CPU}(0, f)$. The energy consumed by the cores throughout the execution of a program is:

$$E_{CPU} = \sum_{k=0}^{\#cores} T_n P_{CPU}(n, f) \tag{3.51}$$

where T_n is the total wall clock time when n cores are active. If the workloads are fully parallelizable, the modeling of T_n can be simplified. Assuming that the program uses n threads, we split the entire execution time of the program, T, into a period during which n cores are active and periods during which no cores are active[1]:

[1]We note that the workloads addressed by us vary less widely than online data-intensive web-services. For more such workloads this assumption results in an underestimation of the power usage, because cores become active at discrete intervals, rather than as a cohort. See Meisner et al. [83] for a detailed power characterization of such workloads.

$$\begin{aligned} E_{CPU} &= T \cdot U_{CPU} \cdot P_{CPU}(n, f) + \\ & \quad T(1 - U_{CPU}) P_{CPU,idle} \end{aligned} \tag{3.52}$$

The power and energy usage of the rest of the system (i.e. video, storage, peripheral devices, voltage stabilizers etc.) is considered fixed and independent of the workload, and accounted in $P_{CPU,idle}$.

When the program is executing on n cores, the CPU utilization will be equal to the ratio of average number of cores doing useful work and memory contention, to the total number of cores:

$$U_{CPU} = \frac{\pi'(m,n) + \pi'(m,n)\omega_\mu(n)}{n} = \frac{\pi(m,n)}{n} \cdot \frac{1 + \omega_\mu}{1 + \omega_\mu + \omega_\sigma} \tag{3.53}$$

The power drawn by a core depends on the type of activity effected by the core. Let $P_{WORK}(n, f)$ be the power consumed by n cores when executing work cycles, and $P_{STALL}(n, f)$ be the power consumed when executing stall cycles. Because threads are considered homogeneous, all cores execute an equal mix of instructions and the power drawn by n cores is the average of P_{WORK} and P_{STALL}, weighted with the ratio of work to stall cycles:

$$P_{CPU} = \frac{w \cdot P_{WORK} + \big(c - w\big) P_{STALL}}{c} \tag{3.54}$$

Both P_{WORK} and P_{STALL} are system characteristics that depend on n and f, while w and $c(n, f)$ are workload characteristics.

The energy incurred by the memory is divided into energy incurred when there are no memory requests, $E_{M,idle}$ and energy incurred when the memory is serving memory requests, $E_{M,active}$:

$$E_M = E_{M,active} + E_{M,idle} \tag{3.55}$$

The total time where requests are serviced by the memory is memory service time, M, while the time when the memory does not service requests is $T - M$. Because $M = r \cdot s_M$:

$$\begin{aligned} E_{M,active} &= M \cdot P_{M,active} \\ E_{M,idle} &= (T - M) P_{M,idle} \end{aligned} \tag{3.56}$$

Similarly, for the I/O requests, the time when the I/O device is busy transferring data is I_T, while the idle time is $T - I_T$:

$$\begin{aligned} E_{I/O} &= E_{I/O,active} + E_{I/O,idle} \\ E_{I/O,active} &= I_T \cdot P_{I/O,active} \\ E_{I/O,idle} &= (T - I_T) \cdot P_{I/O,idle} \end{aligned} \tag{3.57}$$

The model separates the impact of the system parameters from the workload parameters. P_{WORK}, P_{STALL}, $P_{CPU,idle}$, P_M, and $P_{I/O}$ are independent of workloads. Thus, they can be measured once and then used as constants in the model. In contrast, w and I_T depend only on the workload, while c, T, U_{CPU} depend both on workload and on the system, as described in the previous section.

3.3.3 Energy Proportionality

Energy proportionality is a desirable property of a system that consumes power proportional to its useful work output [16]. A perfectly energy proportional system will thus consume zero power when not loaded. In a real system, perfect energy proportionality is not achievable because a hardware device will consume some amount of power even when not loaded, due to circuit design concerns or leakage currents, among other factors.

We define the energy proportionality factor of a device as the ratio of total active energy to total energy spent to execute a program, η:

$$\eta = \frac{E_{active}}{E_{total}} \tag{3.58}$$

Thus, the energy proportionality of the cores, memory and I/O devices are:

$$\eta_{CPU} = \frac{U_{CPU} P_{CPU}(n,f)}{P_{CPU,idle} + U_{CPU}(P_{CPU}(n,f) - P_{CPU,idle})} \tag{3.59}$$

$$\eta_{M} = \frac{M P_{M,active}}{T P_{M,idle} + M(P_{M,idle} - P_{M,active})} \tag{3.60}$$

$$\eta_{I/O} = \frac{I_T P_{I/O,active}}{T P_{I/O,idle} + I_T(P_{I/O,idle} - P_{I/O,active})} \tag{3.61}$$

From the above equations we note that the energy proportionality of a system is maximized when there is no idle time. This implies that the energy proportionality of a system is maximized when there is perfect overlap between the service times incurred by cores, memory and I/O devices:

$$W = M = I \tag{3.62}$$

Thus

$$\frac{w+b}{f\pi(m,n)} = \frac{r_M}{\mu - nL} = \frac{r_I}{B_I} \tag{3.63}$$

We note that by adjusting the core frequency, we can balance the core performance to match the memory performance. We denote with f_k the kneepoint frequency which balances the cores and memory resources such that their response times are equal.

For small values of f, the power cost is small but the execution time is large.

For values of f close but smaller than f_k, the time spent by cores waiting for memory is balanced by performing useful work and the energy-frequency profile reaches the optimal zone. When $f > f_k$, power consumption increases when f increases, but execution time does not decrease, because it is bounded by the response time of the memory requests. This leads to very high energy usage.

Substituting $M(n)$ using equation 3.32 in equation 3.5, we can determine the knee point frequency f as a factor of number of active cores n:

$$f_k(n) = \frac{w+b}{r(n)}(\mu - nL) \tag{3.64}$$

The relationship between f_k and n is linear, but the slope is affected by the arrival rate of memory requests to the memory controller, L, and the ratio of useful work $w + b$ to number of last-level misses r. Equation 3.64 entails that f_k is lower for programs with higher memory contention. This conclusion matches the intuition, because a program with low contention will result in a high knee frequency, thus the program can be executed at high clock frequency without hitting the memory wall. In contrast, a program with high contention will saturate the memory bandwidth even at low clock frequency, if the rate of memory accesses is very high.

Next we discuss the impact of the memory resources and the I/O resources on energy proportionality. From our memory model, we note that when the memory subsystem is close to full utilization, the memory traffic is non-bursty. As such, we can approximate the memory service rate μ with the effective memory bandwidth between the cores and the memory chips, which we denote with B_M. For perfect energy proportionality, the memory and the I/O devices should be fully utilized, and the relationship between the the bottleneck I/O bandwidth and the memory

bandwidth is:

$$B_I = \frac{r_M}{r_I}(B_M - nL) \tag{3.65}$$

Equation 3.65 shows that the I/O throughput per core that fully utilizes the I/O bandwidth drops linearly with the number of cores, but is affected by the memory bandwidth and the number of I/O and memory requests. Unfortunately, most contemporary multicore systems do not allow a dynamic changing of the I/O or of the memory bandwidth. However, even in this case our model can serve as a guide for system architects to provision the off-chip resources for a specific application. For example, in datacenters most multicore systems are used for web-hosting applications that have fairly uniform memory and I/O characteristics. For such applications, our model can serve as a guide for designing custom multicore systems that have the memory and I/O resources specifically tailored to minimize energy wastage. Additionally, we can use the model as a case for suggesting that future energy-efficient multicore systems should be provisioned with the ability to dynamically adjust the I/O and the memory bandwidths, as long as this adjustment results in a lower power usage.

3.4 Summary

This chapter describes our general analytical model for parallelism and energy performance. The general analytical model is split into the parallelism model and power-energy model. The parallelism model predicts the inherent and exploited parallelism of a shared-memory program, and the parallelism loss due to data-dependency, memory contention among cores and I/O overhead. We show an implementation of the general model for commodity multicore systems using the operating system run-queue as the proxy for parallelism, a queueing model for predicting the memory contention factor in multiprocessor UMA and NUMA sys-

Parallelism performance
Inherent program parallelism: $\pi(m,n) = \frac{\sum \pi(m,n,t)\Delta T(n)}{\sum \Delta T(n)}$
Exploited program parallelism: $\pi'(m,n) = \pi(m,n)\frac{1}{\omega_\mu(n)+\omega_\sigma(n)+1}$
Parallelism loss due to data dependency: $\pi_d(m) = m - \pi(m,\infty)$
Parallelism loss due to memory contention: $\pi_\mu(m,n) = \pi(m,n)\frac{\omega_\mu(n)}{\omega_\mu(n)+\omega_\sigma(n)+1}$
Parallelism loss due to I/O overhead: $\pi_\sigma(m,n) = \pi(m,n)\frac{\omega_\sigma(n)}{\omega_\mu(n)+\omega_\sigma(n)+1}$
Energy performance
Average power consumption of a program: $P(n) = P_i + pP_{mc} + \pi(m,n)P_c$
Energy usage of a program: $E(n,f) = E_{CPU} + E_M + E_{I/O}$
Energy proportionality
Knee clock frequency: $f_k(n) = \frac{w+b}{r(n)}(\mu - nL)$
Core energy proportionality: $\eta_{CPU} = \frac{U_{CPU}P_{CPU}(n,f)}{P_{CPU,idle}+U_{CPU}(P_{CPU}(n,f)-P_{CPU,idle})}$
Memory energy proportionality: $\eta_M = \frac{MP_{M,active}}{TP_{M,idle}+M(P_{M,idle}-P_{M,active})}$
I/O energy proportionality: $\eta_{I/O} = \frac{I_T P_{I/O,active}}{TP_{I/O,idle}+I_T(P_{I/O,idle}-P_{I/O,active})}$

Table 3.2: Summary of model equations

tems and a simple I/O performance model. Following our parallelism model, we introduce a model of power and energy requirements of shared-memory programs in multicore systems that can be used to analyze the energy proportionality of a system. Table 3.2 summarizes the equations of our modeling approach.

Chapter 4

Model Parameterization and Validation

In this section we present our experimental parameterization and validation of the general model. First, we discuss the workloads and setup used in our experiments. Second we evaluate the effects of changing the number of active cores on memory contention and on the burstiness of memory traffic. Next, we discuss the model parameterization, during which we measure both the system-dependent and workload-dependent inputs. We perform a sensitivity analysis to select the configuration of the baseline runs that generate the most accurate inputs for our model. Finally we present validation results of the model against measurements on commodity x64 Intel/AMD and low-power ARM Cortex-A9 multicore systems.

4.1 Workloads and Experimental Setup

In this thesis, we focus on workloads spanning different application domains and with diverse degrees of resource requirements. The programs using in the validation and analysis cover high performance computing (HPC), financial computing,

multimedia and web-hosting. We analyze the performance of these workloads on commodity Intel/AMD x64 systems and low-power systems such as ARM Cortex-A9 multicores.

4.1.1 Workloads

We have profiled six shared-memory HPC programs from NPB 3.3 suite [13], two PARSEC 2.1 applications representing financial and multimedia workloads [18] and a program specific to datacenter computing – *memcached* [4]. In the validation we only discuss a subset of the programs, that best illustrate the different cases of core, memory and I/O resource requirements. The programs included in the validation are shown in Table 4.1. In this chapter we present summary results for all programs, and focus the discussion on a smaller set of workloads. In Appendix A we present the detailed validation results over all the programs.

The problem sizes used in the validation are selected such that it always fits in the main memory of the system, without paging out to disk swap. Because our target systems range in main memory size from 1 GB to 64 GB, we discuss for each system the problem size used. In general, we have used large problem sizes that result in executions times of several minutes. For all systems, the resulting working sets are large enough to exceed the sizes of all the caches.

Domain	Name	Parallel kernel
HPC	*EP*	Embarrassingly parallel: low data dependency, low memory
	IS	Parallel sorting: bucket sort of integers
	FT	Spectral methods: fast Fourier transform
	CG	Sparse linear algebra: data with many 0 values
	BT	Dense linear algebra: use matrices and vectors to store data
	SP	Structured grid: penta-diagonal solver
Multimedia	*x264*	Video encoding using H264 codec
Financial	*blackscholes*	European share options pricing using Black-Scholes PDE
Datacenter	*memcached*	In-memory key-value storage and retrieval

Table 4.1: Six NPB 3.3, two PARSEC 2.1 and one datacenter workload

Table 4.1 presents the workloads used during the validation and analysis in this thesis [1].

The HPC workloads are chosen from the NPB 3.3 benchmark [13] that implements HPC dwarfs [10] using OpenMP 2.5. The dwarfs selected scale in terms of both problem size and number of threads, and cover a wide degree of data-dependency and memory contention factors. *EP* is highly parallel with little communication, while *IS*, *FT*, *CG*, *BT* and *SP* have significant memory requirements and data dependency. The programs are compiled using `gcc4.3` with full optimizations (`-O3`) and relaxed floating points options (`-ffast-math`). For the x64 multicore systems we compiled the programs into 64-bit executables, while for the ARM system we have compiled into 32-bit executables using the ARMv7-A instruction set architecture with hard floating point options (`-mfloat-abi=hard -mfpu=neon-vfpv4`).

The financial and multimedia workloads selected from PARSEC 2.1 [18] are implemented using pthreads. The problem size ranges from very small (termed `simsmall`) to very large (termed `native`). These programs have complex data-dependency but typically little memory contention factors and no network I/O requirements. The compile options are similar with the HPC programs.

The datacenter workload is chosen to cover the typical applications run in large datacenters. The most widely used workloads in contemporary datacenters are webservers such as *Apache* or *lighttpd* and in memory cache programs such as *memcached* [82, 83]. Because the webservers have complex storage I/O requirements, we have opted to use *memcached* version 1.4.13 as our representative datacenter workload. *Memcached* is an in-memory key-value store that is widely used to serve dynamic page content by web giants such as Facebook, Twitter and

[1] In this thesis, we use the NPB notation for workloads. For example *EP.C* refers to program *EP* with input *C*. When a program is referred without input, it refers to a characteristic of the program that is input-independent.

Amazon, among others. By relying on an in-memory cache, they significantly reduce the need to access the storage-backed databases, thus improving the response time of their requests. We ran *memcached* on a cache size of 250 MB (the cache size is chosen such that it fits even in systems with only 1 GB of RAM). Memcached uses pthreads as worker threads to serve requests. The request are issued by program *memslap*[2] version 0.44. *Memslap* is run on different system, which is connected to the system under test through an Ethernet network interface. *Memslap* continuously sends requests to *memcached*, with the thinking time between requests independent on the response rate of the request. The datacenter workload *memcached* has complex CPU, memory and I/O requirements, but negligible data-dependency among the worker threads.

4.1.2 Systems

The measurements and the validation are conducted over a range of traditional (x64) and low-power ARM multicore systems:

1. **Intel UMA (8 cores)**: Dual socket Intel E5320 at 1.87 GHz, 4 cores and 8 MB L2 cache per socket, 1 memory controller with 4 GB dual channel DDR2 RAM, 1 Gbps Ethernet, Linux 2.6.22 64-bit;

2. **Intel NUMA (24 cores)**: Dual socket Intel X5650 at 2.67 GHz, 6 cores with 12 hardware threads and 12 MB L3 cache per socket, 2 memory controllers with 24 GB RAM single channel DDR3 RAM, 1 Gbps Ethernet, Linux 2.6.35 64-bit;

3. **AMD NUMA (48 cores)**: Quad socket AMD Opteron 6172 at 2.10 GHz, 12 cores with 10MB L3 cache per socket, eight memory controllers with 64

[2]Memslap issues requests with constant size and uniform popularity, which may lead to higher CPU utilization than in actual usage of memcached. For practical traffic characteristics see Atikoglu et al. [11]

GB dual-channel DDR3 RAM, 1 Gbps Ethernet, Linux 2.6.35 64-bit;

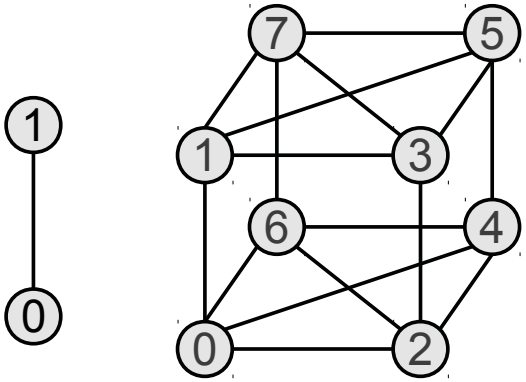
4. **ARM Cortex-A9 (4 cores)**: Samsung Exynos 4412 System-on-a-Chip with 4 ARM Cortex-A9 cores ranging from 0.2 to 1.4 GHz, 1 MB L2 cache, one memory controller with 1 GB dual-channel LPDDR2, 100 Mbps Ethernet, Linux 3.6.0 32-bit.

In systems with simultaneous multithreading, we consider the two hardware threads of each physical core as logical cores, because the objective of this study is off-chip memory requests. Each of the two hardware threads issue memory requests independently, so from the perspective of the memory accesses, the physical core with two hardware threads appears as two cores. Therefore, we consider Intel NUMA as having 24 cores.

The multicore systems used in experiments have two main types of memory architectures, uniform memory access (UMA) and non-uniform memory access (NUMA). In our UMA systems, two quad-core processors are connected to a common memory controller through private buses. The last-level cache in UMA is semi-unified, because a pair of cores shares a common L2 cache. Since all the cores share one memory controller, contention occurs when memory requests exceed the capacity of the memory controller. In contrast, each multicore processor in a NUMA system accesses its own memory through its dedicated local memory controller. A core accesses memory owned by another processor through its inter-processor connection network. All multiprocessor systems from Intel since the *Nehalem* microarchitectural generation (2009) and from AMD since the *K8* generation (2003) are NUMA systems.

The interconnect networks for the NUMA systems is shown in Figure 4.1. Intel NUMA has two memory controllers directly interconnected, therefore there are two latencies for accessing the memory – direct and one hop. AMD NUMA

has eight memory controllers interconnected through a partial mesh, and there are three latencies of accessing the memory – direct, one hop and two hops.



(a) Intel NUMA (b) AMD NUMA

Figure 4.1: Memory interconnect of NUMA systems

The program was partitioned into a fixed number of threads. The trace of the operating system run-queue was obtained using a C program that samples the `procfs` entries to log the number of runnable threads and their user-level CPU service time. We used `time` system utility to measure the wall clock time, `sched_setaffinity` system call to restrict the number of cores allocated to a program, `numactl` utility to specify the memory access nodes.

The number of cores was varied from one to maximum number of cores of the machine using a *fill-processor-first* policy. For Intel NUMA, memory controller 0 was used until all cores from processor 0 were active, and then memory controller 1 was activated. For AMD NUMA, the memory controllers were activated in order of increasing latency: $0, 2, 4, 6, 1, 3, 5, 7$. We used PAPI 4.1.2 to measure the following counters: `PAPI_TOT_CYC` for the number of cycles, `PAPI_TOT_INS` for the number of instruction, `PAPI_RES_STL` for stall cycles, `PAPI_L2_TCM` for the number of cache misses, `LLC_MISSES` on Intel NUMA and `L3_CACHE_MISSES` on AMD NUMA for L3 misses. The work cycles were determined as the difference between all cycles and stall cycles. We used `papiex` tool to measure the hardware

counters of the profiled applications only, without interference from background processes and operating system. To ensure that the memory bandwidth is not shared with any other process, we turned off all non-essential processes and we run the profiled application with the highest priority allowed (process nice value -20). Unless otherwise stated, we run each experiment five times, and for the cases where we found significant differences among the runs, we present and discuss the relative difference among the runs.

4.1.3 Power and Energy Measurement

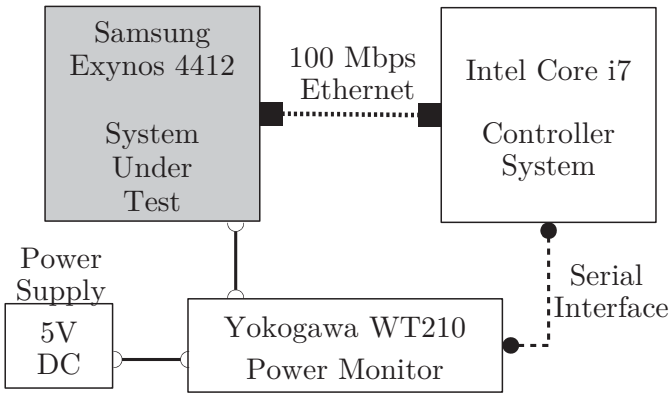


Figure 4.2: Power and energy measurement setup

For measuring the power and energy consumption, we use a Yokogawa WT210 digital power meter configured to measure the power and energy per one input channel. Because we measure the overall system power, we connect the power meter to the power supply of the entire system. We perform power and energy measurements only for the low-power computing device, because we can measure more accurately the DC power directly supplied to the board. For Intel and AMD systems, the AC power supplied to the system is converted to DC using an internal power supply. Due to this conversion from AC to DC, the sensitivity of the measurements are much lower and thus we cannot accurately detect the changes in power and energy usage, when the programs exert different types of

service requirements. For these reasons, we have performed validation of the power and energy model only on the ARM system.

Figure 4.2 shows the setup of the power measuring system. An Intel Core i7 system is used as a controller, connected to the ARM system under test using a 100 Mbps Ethernet link. The controller starts and stops all experiments and collects all the data. The power monitor outputs every second the average power during the last second, and total energy used since an arbitrary time moment.

4.2 Measurement Analysis of Memory Contention

This section presents our observations on the memory contention in large multicore systems with different memory architectures. We first show the effects of memory contention on parallel programs by varying the number of active cores for different problem sizes. Next, we study the nature of memory contention by profiling the patterns of memory access.

4.2.1 Impact of Number of Cores on Memory Contention

To understand the impact of the number of active cores on memory contention, we profile a suite of HPC programs and PARSEC applications, by changing the number of cores and memory placement policies. We evaluate the growth in the work of the program, for different problem sizes.

Experimental Setup Our measurement experiments focus on understanding how the execution of a program is affected by the off-chip memory traffic. We first perform a set of measurements on the number of cycles required to execute the programs when using different number of cores. Second we measure the patterns of memory traffic to understand the nature of the memory contention. Because we

evaluate the performance of the memory subsystem, we use the HPC workloads, which are much more memory-intensive compared to the PARSEC workloads.

Memory Contention vs. Number of Cores For each program, we measure the *total number of cycles* required to execute the program including initialization and cleanup, as well as the number of stall and work cycles, and the total number of last-level cache misses. Mainstream processor cores are based on superscalar and deeply pipelined microarchitectures. Thus, in each cycle a core can execute multiple integer and floating point operations and issue multiple memory requests. If the operands of an instruction are available in the registers, the execution of the instruction can proceed. Otherwise, the core stores the instruction in the *instruction dispatch queue* until the operands are fetched from the first-level cache. If the data is not available in the first-level cache, then it attempts to fetch it from the subsequent levels of cache. If the data is not found in cache, the core issues a *memory request* to the main memory. Due to the long latencies of accessing higher levels of cache or the main memory, instructions can be stopped for several hundreds of cycles [62]. If the entire dispatch queue is filled with instructions waiting for data, no instructions can proceed and the core is stalled waiting for memory. If no operations are completed during a cycle, it is called a *stall cycle*. In contrast, if at least one instructions is completed during the cycle then is termed a *work cycle*. Next we discuss our observations.

Table 4.2 shows the normalized increase in the total number of cycles for five HPC dwarfs with small (W) and large (C) problem size[3]. The increase in the number of cycles is defined as the difference between the total cycles incurred using n cores and one core, normalized to the number of cycles on one core. We present the normalized increase for n equal to half and all cores of the systems (i.e.

[3]Problem sizes are denoted by letters, and are according to NPB benchmark specification. Notation CG.C means program CG problem size C.

Program	Size	Normalized Increase in Number of Cycles					
		Intel UMA		Intel NUMA		AMD NUMA	
		#Cores		#Cores		#Cores	
		n=4	n=8	n=12	n=24	n=24	n=48
EP	*W*	0.00	0.00	0.03	0.57	0.01	0.59
IS		0.35	0.90	0.33	0.33	0.21	0.44
FT		0.41	1.04	0.18	0.34	0.11	0.23
CG		0.06	0.04	0.10	0.43	0.11	0.13
BT		0.14	0.33	0.08	0.30	0.11	0.12
SP		0.27	0.86	0.10	0.50	0.13	0.21
EP	*C*	0.00	0.00	0.01	0.54	0.06	0.55
IS		0.14	0.56	0.26	0.85	0.40	0.70
FT		0.70	1.76	1.62	3.94	0.39	0.46
CG		0.64	2.41	1.43	3.31	0.83	1.91
BT		0.48	1.18	1.73	2.25	0.20	0.75
SP		3.14	7.04	6.55	11.59	4.69	9.84

Table 4.2: Normalized increase in number of cycles in HPC dwarfs

4 and 8 on Intel UMA, 12 and 24 on Intel NUMA, 24 and 48 on AMD NUMA). Because FT.C working set size exceeds 4 GB and leads to swapping in our Intel UMA system, we use class B as large problem size for program FT on Intel UMA. Overall, on all three systems the increase in number of cycles is more pronounced for higher number of active cores.

We identified two main types of behavior with respect to the number of active cores:

1. Programs with small problem size or working sets which are cached effectively generate low number of off-chip requests. This leads to a negligible growth in number of cycles when the number of active cores increases.

2. Programs with large problem sizes generate high number of off-chip memory requests which lead to a significant growth in the number of cycles when the number of cores increase.

We show these two patterns using a representative HPC program. Program *CG* is a parallel application that approximates the largest eigenvalues for a large

and sparse matrix. We use a small and a large problem size [13]:

1. Class W consists of a matrix with $7,000^2$ elements;

2. Class C consists of a matrix with $150,000^2$ elements.

CG is representative for all HPC applications and is chosen because it represents a case with moderate memory contention (SP has higher contention, FT, IS, EP, and all PARSEC programs have lower contention). Small problem size W generates a small increase in the number of cycles, even on large number of cores. The largest increase for problem $CG.W$ is reached on Intel NUMA on 24 cores, with 63% increase. In contrast, $CG.C$ shows a large growth in number of cycles, on all three systems, with a maximum increase of 331% on Intel NUMA.

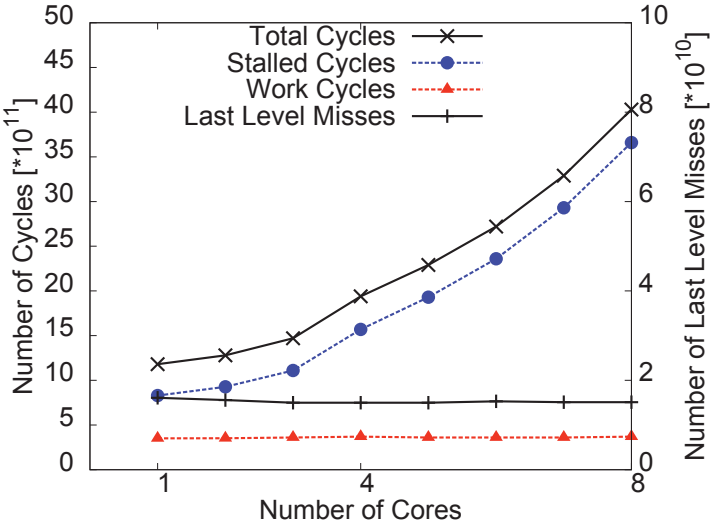
Next, we focus the discussion on the more interesting case of large problem size. Figure 4.3 shows the results for $CG.C$. On all systems, there are three main observations when the number of active cores is increased:

1. The number of total cycles increases non-uniformly.

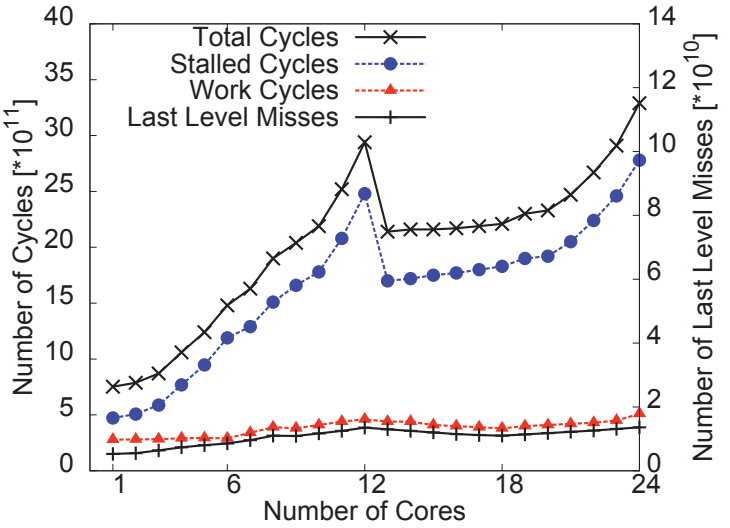
2. The growth in number of total cycles is due to an increase in number of stall cycles.

3. The number of work cycles and the number of last level cache misses grow insignificantly.

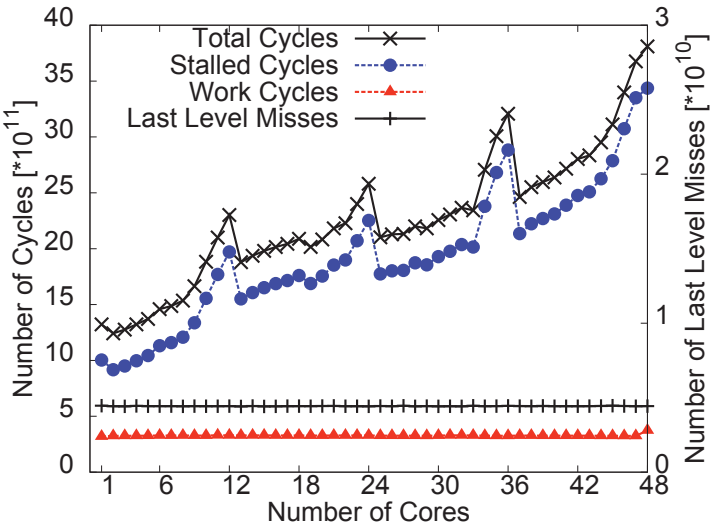
For problem size C, the patterns of growth depend on the architecture and the number of memory controllers. For Intel UMA we observe two sustained growth intervals, the first from one to four cores, the second from five to eight. This corresponds to a per-processor pattern of growth. Similarly, on Intel NUMA, the growth on the first processor (1 to 12 cores) is similar in shape with the growth from 13 to 24 cores. However, when the second processor is activated (from 13



(a) Intel UMA (Xeon E5320)



(b) Intel NUMA (Xeon X5650)



(c) AMD NUMA (Opteron 6172)

Figure 4.3: Effect of varying the number of cores on *CG.C*

cores onward), there is a small decrease in memory contention which results from the added memory bandwidth of the second memory controller. On AMD NUMA, there are four intervals of growth, each corresponding to a processor. Although each processor has two memory controllers, which are activated at 6, 18, 30 and 42 active cores, their activation does not change significantly the shape of the growth.

As the number of cores is increased, we observe that the number of work cycles remains roughly constant because the critical path of the program is dominated by instruction waiting for operands fetched from memory. Instruction execution is interleaved almost fully with fetching operands. Thus, the increase in the total number of cycles is dominated by waiting for memory requests or stall cycles. Table 4.3 shows presents experimental evidence for the assumption that work cycles and last-level cache misses do not change significantly when the number of active cores change. We measure $w(n)$ and $r(n)$ and observe that on Intel UMA, the number of $r(n)$ does not change significantly neither among different runs nor when changing the number of cores. On Intel and AMD NUMA, the $r(n)$ increases slightly with n, and therefore, we use a normalized value of $w(n)$ to $r(n)$. We computed the coefficient of variation of $w(n)$ and $r(n)$ for UMA and of $\frac{w(n)}{r(n)}$ for NUMA. For NUMA, we restricted the memory access to the local controller only. Overall, the variation of $w(n)$ to $r(n)$ is very small, confirming our assumption

System	**Coefficient of variation**	**Programs**					
		EP.C	**BT.C**	**SP.C**	**FT.B**	**IS.C**	**CG.C**
Intel UMA (Xeon E5320)	$w(n)$	0.00	0.00	0.04	0.02	0.01	0.02
	$r_M(n)$	1.30	0.02	0.11	0.06	0.00	0.03
Intel NUMA (Xeon E5520)	$\frac{w(n)}{r_M(n)}$	3.30	0.03	0.06	0.02	0.04	0.83

Table 4.3: Variation of $r_M(n)$ and $w(n)$ using one memory controller

for weak-scaling programs. This can be clearly seen in Figure 4.3, as the shape of growth of the stall cycles closely follows the shape of growth of total cycles.

Another interesting observation is that the number of last-level cache misses, L2 for UMA and L3 for NUMA, remains stable. Because we fixed the number of threads, and varied only the number of cores, the total number of instructions also remains constant for a given problem size. This confirms that the increase in the total number of cycles is the result of contention for off-chip memory requests, rather than an increase in the number of memory requests or an increase in the number of instructions executed.

4.2.2 Burstiness of Memory Traffic

To understand the nature of memory contention, we profiled the memory access patterns. Using a very fine grained sampler we have developed, we measure the number of last-level cache misses that occur every five microseconds. This allows us to determine the burstiness of memory accesses over time. The sample size of five microseconds gives very good resolution of the lifetime of the applications, but has minimal impact on intrusiveness. The difference between the number of last level cache misses with and without the profiler is less than 3%.

The second objective of our experiments is to analyze the relationship between problem size and the burstiness of memory traffic. To study this, we measure the burstiness of last-level cache misses over time. Figures 4.4 and 4.5 show the burstiness of off-chip memory traffic for two representative programs *CG* and *x264*, each for a selection of problem size ranging from small and large, as shown in Table 4.4. Program *CG* determines the largest eigenvalues of a sparse matrix, while *x264* performs H.264 video encoding for different frame numbers and resolutions.

Figures 4.4 and 4.5 show the burstiness of the memory traffic for both programs, on Intel NUMA using 24 threads and 24 cores. The graph in log-log scale plots $P(Burst\ Size > x)$, the probability that the memory burst size exceeds the

Program and Size	Problem Size Description
CG.S	matrix of size $1,400^2$
CG.W	matrix of size $7,000^2$
CG.A	matrix of size $14,000^2$
CG.B	matrix of size $75,000^2$
CG.C	matrix of size $150,000^2$
x264.simsmall	8 frames at 640 x 360
x264.simmedium	32 frames at 640 x 360
x264.simlarge	128 frames at 640 x 360
x264.native	512 frames at 1,920 x 1,080

Table 4.4: Problem size description for *CG* and *x264*

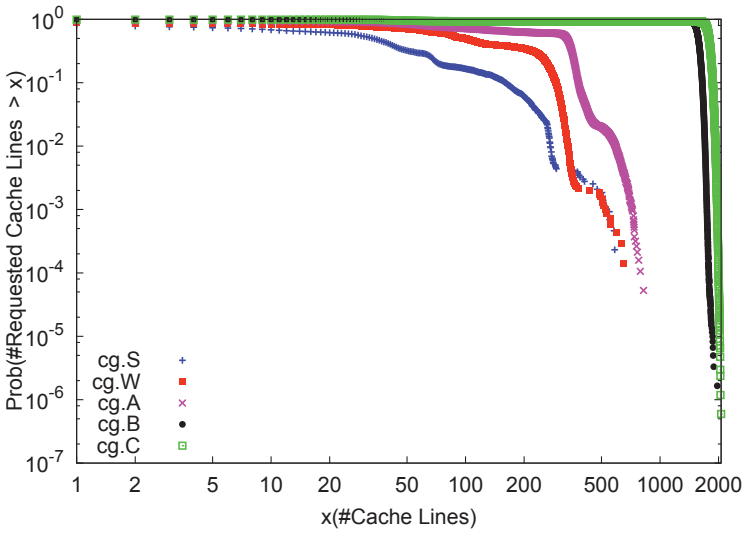


Figure 4.4: Burstiness of off-chip memory traffic: HPC dwarf *CG*

number of cache lines x, for different sizes of cache lines. The plot shows that the size of memory requests varies widely, ranging from four to seven orders of magnitude from small to large program sizes. However, the small (S and W for *CG*, and simsmall, simmedium and simlarge for *x264*) and large problem sizes (B and C for CG, native for x264) behave quite differently. In small problem size, for both programs the long tail property of the distribution of burst size is prominent. For bursts larger than 50 cache lines, $log\ P(BurstSize > x)$ decreases linearly with $log\ x$ with the log of burst size in approximately a diagonal straight line. This confirms that the traffic is highly bursty, which is in line with previous

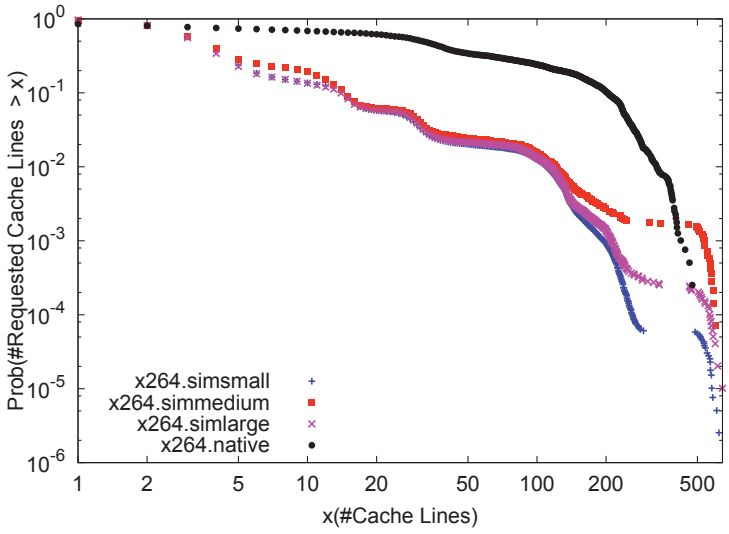


Figure 4.5: Burstiness of off-chip memory traffic: PARSEC *x264*

observations about the nature of memory traffic [63]. However, as the problem size increases, the deviation from a decreasing diagonal line becomes clearer, and for large problem sizes B and C in program *CG* the long tail property is absent. This means that *CG* memory traffic is not significantly bursty. The intuitive explanation behind this observation is that large problem size B and C the memory bandwidth is highly utilized and therefore there are no significant time intervals without memory requests. The same trend of decreasing burstiness when problem size increases was observed for all programs with significant memory contention

(IS, CG, FT, BT and SP). The results on the other two systems, Intel UMA and AMD NUMA are roughly similar.

In conclusion, our experiments show two types of memory contention behavior with respect to problem size:

1. Small problem sizes lead to small contention for off-chip resources but result in highly bursty traffic.

2. Large problem sizes can lead to non-bursty memory traffic but results in large off-chip memory contention among cores.

From these results we conclude that we can apply a stochastic single-server queueing model to analyze the memory response time for large parallel programs on multicore systems, as proposed in section 3.2.5. In the model parameterization section we show that the program with bursty memory traffic can be modeled using a Pareto distribution of service time of memory requests, while the non-bursty memory traffic can be modeled using an exponential distribution of service time.

4.3 Model Parametrization

In this section we discuss the selection of two types of model inputs: workload-specific and system-specific. Workload specific inputs are selected using a series of baseline runs on configurations determined using a sensitivity analysis. System-specific parameters are measured only once per system.

4.3.1 Baseline Runs Configuration

We discuss the configuration of the baseline run by sampling the run-queue size and exemplify this selection on program $BT.C$.

The choice of run-queue sample interval is important in determining the correct parallelism profile, and we consider two opposing aspects:

1. If the sample interval is large, there is a higher probability of not detecting parallelism changes within the interval. Ideally, the sample interval must thus be lower than the time between two consecutive changes in the value of $\pi(m, \infty, t)$.

2. It the sample interval is too small, the service time of the threads cannot be accurately determined, leading to incorrect compensation for load imbalances.

Quantitative analysis is performed to determine the optimal *run-queue sample intervals* and *number of cores* for the baseline run. Figure 4.6 shows the modeled $\pi(m, \infty)$ for program BT class C (BT.C), partitioned in 8 threads, with baselines conducted on 1, 2, 4 and 8 cores on the Intel UMA system. Figure 4.6 shows that

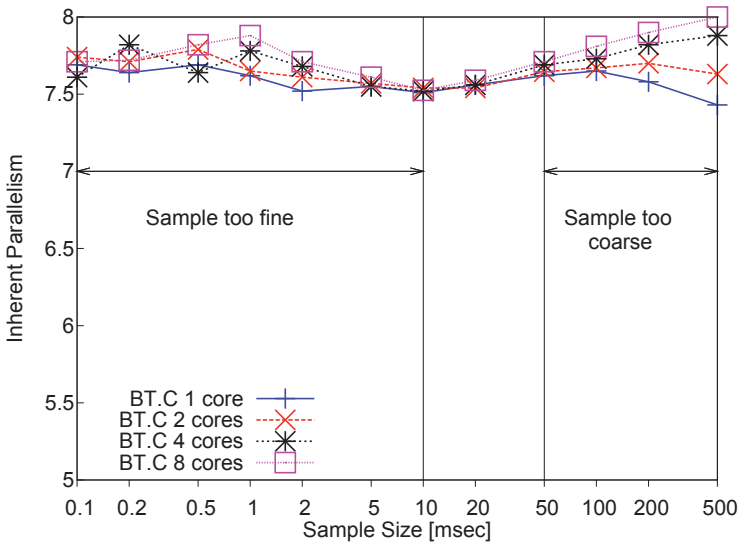


Figure 4.6: Modeled inherent parallelism: effect of run-queue sample interval

sample intervals lower than 10 ms are impractical, because the *procfs* updates the CPU service time of the threads in 10 ms intervals. Because on our systems the

scheduler quanta ranges between around 100 ms for Intel UMA and 80 ms for Intel NUMA, the useful interval of sampling is between 10-80 ms.

The measured inherent parallelism of BT.C on UMA, for $m = 8$ threads, $\pi(8, 8)$ is 7.70, averaged over 10 runs. Based on this analysis, we conclude that the accuracy of our model depends on the relative difference between sample intervals and threads inter-synchronization time. We therefore opt for a value of 10 ms, the smallest in the useful range. Furthermore, the number of cores in the baseline run does not change the predicted value of $\pi(m, \infty)$ significantly. Therefore we select *one core* as the value for the baseline runs, to capture the largest number of samples.

4.3.2 Workload Parameterization

Based on the observations from the memory contention measurement experiments, we classify the workloads into two categories:

1. Programs with low memory contention, which trigger few off-chip requests are considered to be CPU-bounded on all configurations. Thus, the response times of the memory requests is totally overlapped with the response time of the core service requests. When applying the model, we consider the response time of the memory requests negligible, when compared to the larger response time of the core service requests.

2. Programs with high memory contention trigger sufficient off-chip memory requests to become memory-bounded. We use a M/M/1 queueing model to predict the response time of the program as a factor of number of active cores. These programs can become memory-bounded on configurations with large core counts or high core frequencies.

System	Program	Growth $\frac{C(n_{max})}{C(1)}$	Bottleneck	$\frac{\mu}{r_M}[\times 10^{13}]$	$\frac{L}{r_M}[\times 10^{13}]$
Intel UMA	*EP.C*	0.99	Cores	–	–
	IS.C	1.56	Memory	36.73	898.8
	CG.C	3.42	Memory	11.07	95.81
	FT.B	2.76	Memory	61.76	510.19
	BT.C	2.18	Memory	2.63	26.9
	SP.C	8.04	Memory	6.76	33.93
	x264	1.10	Cores	–	–
	blackscholes	1.02	Cores	–	–
	memcached	1.55	I/O	–	–
Intel NUMA	*EP.C*	1.02	Cores	–	–
	IS.C	1.27	Memory	1031.43	19.29
	CG.C	2.44	Memory	139.74	7.12
	FT.B	2.62	Memory	127.10	6.77
	BT.C	1.74	Memory	109.31	11.33
	SP.C	7.54	Memory	48.16	3.52
	x264	1.21	Cores	–	–
	blackscholes	1.00	Cores	–	–
	memcached	1.36	I/O	–	–
AMD NUMA	*EP.C*	1.30	Cores	–	–
	IS.C	1.32	Memory	738.01	8.08
	CG.C	1.72	Memory	78.69	2.93
	FT.C	1.67	Memory	116.34	2.57
	BT.C	1.2	Memory	228.58	3.35
	SP.C	2.88	Memory	318.46	17.36
	x264	1.422	Cores	–	–
	blackscholes	1.31	Cores	–	–
	memcached	1.65	I/O	–	–
ARM Cortex-A9	*EP.C*	1.01	Cores	–	–
	IS.B	2.29	Memory	13.73	86.87
	CG.B	2.60	Memory	20.42	119.90
	FT.A	3.11	Memory	84.88	460.40
	BT.C	1.90	Memory	12.29	90.18
	SP.C	2.67	Memory	4.23	24.15
	x264	1.17	Cores	–	–
	blackscholes	1.08	Cores	–	–
	memcached	0.64	I/O	–	–

Table 4.5: Parameterization of memory contention model

Table 4.5 shows the classification of each workload based on the system bottleneck such as CPU, memory or I/O. For each program, we report the normalized increased in the total cycles incurred when using one core against total cycles incurred when executing on the total number of cores that fully exercises one memory controller. For example, on the Intel UMA, since all the processors share the main memory, $n_{max} = 8$ is the maximum number of cores that share one memory domain. For ARM Cortex-A9, $n_{max} = 4$. For the NUMA systems, we use $n_{max} = 12$ on Intel NUMA and $n_{max} = 12$ on AMD NUMA.

For memory-bounded programs, we provide the empirical parameters μ and L for the M/M/1 queueing models that describe the response time of the memory requests. Since for our weak-scaling programs, the number of last level cache misses r_M is assumed constant, we normalize the values of μ and L to the total last level cache misses r_M.

4.3.3 System Parameterization

Freq. [GHz]	Idle Power [mW]		Stall cycles power [mW]				Float power [mW]				Integer power [mW]			
	System	Processor	Number of Cores				Number of Cores				Number of Cores			
			1	2	3	4	1	2	3	4	1	2	3	4
0.2	1,740	1,512	–	–	–	21	35	70	108	137	43	90	132	176
0.3	1,750	1,522	–	–	34	66	51	105	159	212	63	132	196	270
0.4	1,762	1,534	–	45	88	131	73	149	228	289	92	188	280	360
0.5	1,778	1,550	–	90	142	194	97	200	292	372	124	249	352	467
0.6	1,796	1,568	22	128	192	255	122	244	360	465	152	306	436	592
0.7	1,811	1,583	46	167	237	309	149	291	430	569	184	358	522	734
0.8	1,823	1,595	74	209	284	366	176	342	542	687	218	419	647	859
0.9	1,850	1,622	108	248	346	438	212	403	643	825	256	548	780	1,028
1.0	1,880	1,652	150	308	428	583	255	540	779	1,005	310	655	941	1,260
1.1	1,908	1,680	174	356	537	672	302	642	922	1,214	370	777	1,152	1,532
1.2	1,952	1,724	218	476	631	799	366	774	1,123	1,485	488	944	1,391	1,868
1.3	2,026	1,798	269	563	752	963	472	933	1,377	1,854	574	1,149	1,749	2,384
1.4	2,081	1,853	369	649	871	1,114	550	1,079	1,609	2,220	662	1,328	2,079	2,869

Table 4.6: Static power characterization of ARM Cortex-A9 (Exynos 4412)

To determine the system parameters used as model inputs by our model, we execute a series of microbenchmarks designed by us to determine the power consumption for different types of activities. Each of the three programs stresses one type of CPU activity: work integer cycles, work floating point cycles and stall

cycles due to waiting for memory. Additionally, we collect the idle power. To determine the power drawn by memory and Ethernet device, we selectively turn on and off these components. All the power values reported in this section are obtained by averaging the results across three repetitions. Table 4.6 shows the static power characteristics of the system.

First we determine the total system power under idle load, when changing the core frequency. We measured total system power and processor-only power, which is obtained by discounting the power drawn by the memory and I/O components. Thus, in our analysis the total processor power includes the power consumed by miscellaneous system components such as GPU, peripherals, voltage converters and stabilizers and other motherboard circuitry. The power drawn by these components is considered fixed and independent of the workloads. Next, we profile the processor active power when the cores execute two types of work cycles: integer operations and floating point operations. To determine this power, we designed a microbenchmark that achieves close to 100% core pipeline utilization under each type of operations. The results are determined for different number of cores, under each supported clock frequency. To profile the stall cycles, we designed a microbenchmark that reads a large amount of data from memory, and continuously attempting to miss the last level of cache. This benchmark trigger more than 90% stall cycles in the cores pipeline and intense memory activity. For this microbenchmark, we measure the total system power and deduct the power incurred by the memory. As shown in the table, for small core frequencies and core counts, the processor does not fully stress the memory bandwidth, and thus, the stall cycles power cannot be measured for these configurations. When applying the model, we approximate by zero the power drawn on these configurations when the cores are executing stall cycles only. The memory idle power is taken from the literature [80] and is approximated as 28 mW, which is typical for a LPDDR2

DRAM module powered at 1.2 Volts. We measure the active memory power by running the microbenchmark that constantly misses the cache. Under this state, the memory draws approximately 248 mW, derived after discounting the power drawn by the rest of the system. The JEDEC standard for LPPDR2 DRAM indicates a specification of 250 mW power consumption when powered at 1.2 V [3], which confirms the parametrization results. The I/O power load is determined by measuring the total power with network card under full load, under idle load and turned off. The Ethernet card draws 200 mW, irrespective of the load.

A power measurement experiment on a Samsung Galaxy S3 phone done using direct measurement with attached circuit probes obtains similar figures for CPU and memory power consumption [23]. Considering that the Samsung Galaxy S3 and the Exynos 4412 low-power server analyzed by us have identical CPU and memory subsystem, this stands to confirm the accuracy of our static power characterization.

4.4 Models Validation

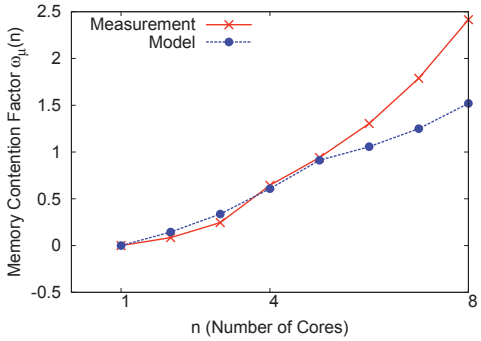
In this section, we discuss the validation of our parallelism model against measurements. First we validate the memory and I/O contention models individually. Next we show the validation of the entire models for inherent and exploited parallelism. For each type of validation, we show in this chapter a subset of the results, focusing on programs that exhibit stronger resource demands. For the memory contention model we use six HPC dwarfs from the NPB benchmark suite that cover a wide range of the intensity of CPU and memory service-demands. From PARSEC, we chose *x264* and *blackscholes*, which are programs targeting multimedia and financial analysis. Finally, as representative for datacenter server workloads, we use *memcached*, which can exhibit complex CPU, memory and I/O

demands. For each model we always present summary results for the six dwarfs, the two PARSEC programs and memcached, but discuss in detail the validation for only a subset of programs, discussed in each validation subsection. The detailed validation results for all programs are shown in Appendix A.

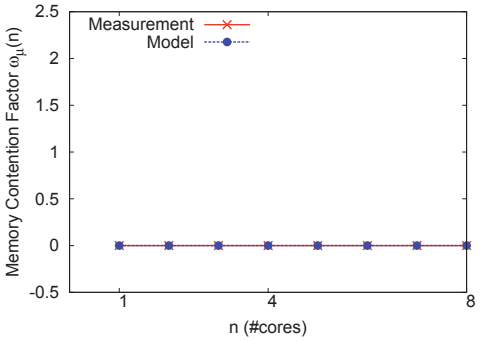
4.4.1 Memory Contention

We discuss the validation and accuracy of our model against measurements before analyzing the effects of varying the number of cores. We show detailed validation results on a program with large contention, *CG.C* and another with small contention, *EP.C*. We also provide summary validation results for all HPC dwarfs, two PARSEC application and *memcached*.

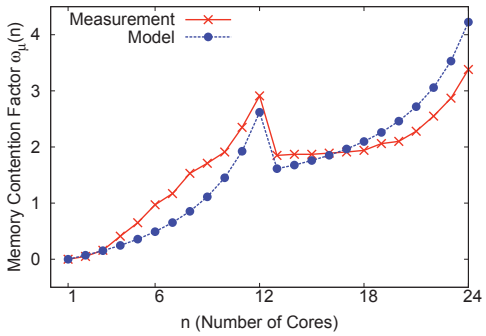
Figure 4.7(a), (c) and (e) shows the comparison between modeled and measured degree of memory contention for program *CG.C* using a fill-processor-first-policy on the traditional multicore system. The average relative error across all measured and predicted model results of memory contention factor ω_μ is 5-11% on all three systems. For Intel UMA we use three measured values of $c(n)$ to apply the model: $c(1)$, $c(4)$ and $c(5)$ and achieve average accuracy of 6%. For AMD NUMA, we use five measured values as inputs: $c(1)$, $c(12)$, $c(13)$, $c(25)$ and $c(37)$ and achieve the best accuracy with error less than 5% across all problems with large contention. For AMD NUMA, we could use three values, $c(1)$, $c(12)$, $c(13)$ and assume that all interconnects are homogeneous, but this degrades the prediction accuracy up to 25% average relative error. On Intel NUMA, we use three measured values of $c(n)$: $c(1)$, $c(12)$ and $c(13)$ and the model reaches the lowest accuracy with 11% average error, largely due to the misprediction around the values of $c(6)$ and $c(18)$. This misprediction is caused by two factors: (i) oversubscription effects and (ii) variability of measurement values. Because we fix the number of threads to 24 on Intel NUMA but vary the number of cores, there will be more than



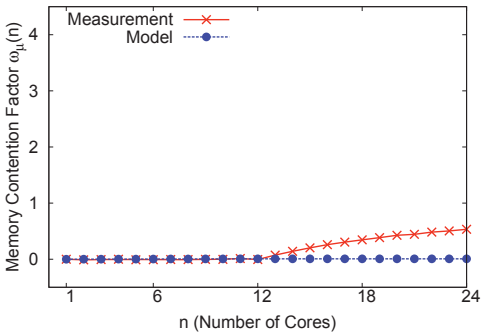
(a) *CG.C* on Intel UMA (Xeon E5320)



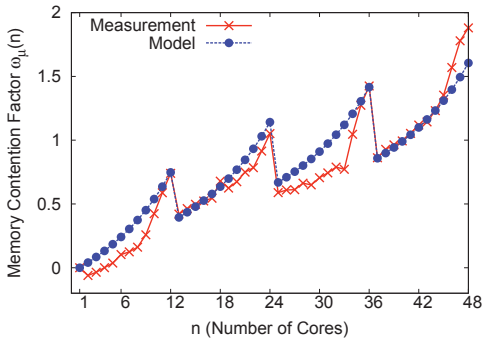
(b) *EP.C* on Intel UMA (Xeon E5320)



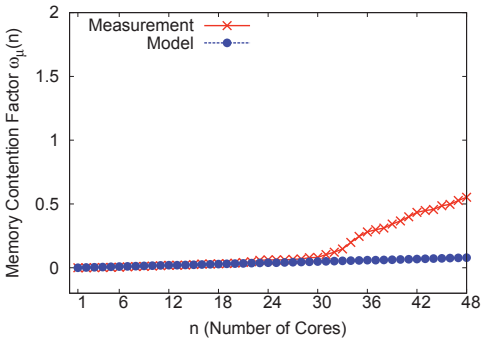
(c) *CG.C* on Intel NUMA (Xeon X5650)



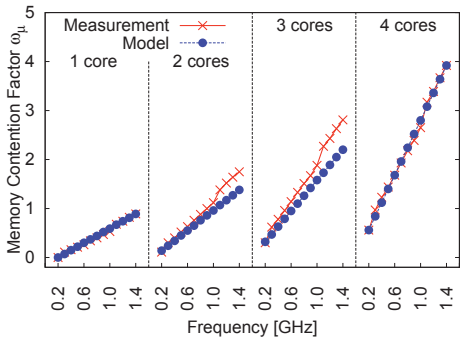
(d) *EP.C* on Intel NUMA (Xeon X5650)



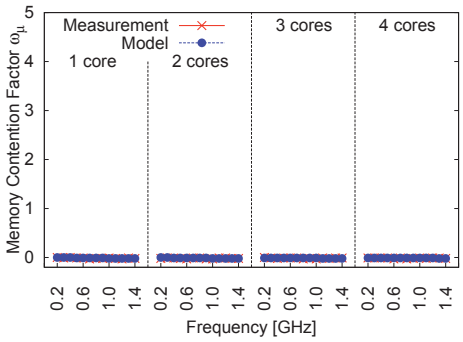
(e) *CG.C* on AMD NUMA (Opteron 6172)



(f) *EP.C* on AMD NUMA (Opteron 6172)



(g) *CG.B* on ARM Cortex-A9 (Exynos 4412)



(h) *EP.C* on ARM Cortex-A9 (Exynos 4412)

Figure 4.7: Validation of memory contention model: *CG* and *EP*

one thread executing on each core. Furthermore, to counter the variability of measurement values due to the operating system scheduler moving threads across NUMA domains, we bind each thread to a specific core. This has reduced the variability of the results, but has introduced negative caching effects between the threads that share the same core. These effects are more pronounced when the oversubscription factor (ratio of threads to cores) is large or a round number, which largely corresponds with the effects observed in related work [55]. The patterns of growth observed on the other programs with large contention (FT and SP) were similar to those observed on CG. On all three systems, the core clock is fixed to the maximum supported frequency.

Figure 4.7 (g) and (h) show the validation of the memory contention model on the low-power ARM Cortex-A9 system. The two graphs show the effects of both changing the number of cores and the core frequency. In order to apply the model, we perform four baseline runs. The first two baseline runs are performed on one core at the minimum frequency ($f = 200$ GHz) and at maximum frequency ($f = 1400$ GHz). The last two baseline runs are performed using all four cores at minimum frequency at maximum frequency, respectively. We apply the M/M/1 queueing model and regress the number of cycles incurred as the number of cores increases, for both $f = 200$ MHz and $f = 1400$ MHz. This allows us to determine the cycles incurred on all four cores when operating at the two extreme frequency points. To predict for intermediate frequency points, we leverage the linear increase in number of memory cycles as the core frequency increases, as described by Equation 3.29. Thus, using the four baseline configuration we can predict the number of cycles for all 52 configurations (4 cores $\times$ 13 frequency points).

Programs with low contention do not result in an significant increase in number of cycles when the number of active cores is increased. Figure 4.7(b), (d), (f) and (h) show the modeled and measured values of contention for $EP.C$. For UMA,

memory contention is negligible because demand for memory that arises from more cores can be met by the cache and off-chip memory resources. However, the NUMA architectures show two interesting trends. The effect of positive memory contention ($\omega_\mu(n) < 0$) is observed with less than 11 cores on $EP.C$ running on Intel NUMA because adding cores also increases memory resources ($L1$ and $L2$ cache). Beyond one processor, memory contention increases to 50%, which is not captured by our model. This is caused by an increase in number of last level cache misses, from $1,800$ misses on one core to 31 millions on 24 cores. Our model assumes the number of work cycles and last level misses constant. This assumptions holds for programs with large memory contention, but may not be for programs with low contention, such as EP. Furthermore, the increase in degree of contention is correlated with the latency of memory accesses. On AMD NUMA, from 1 to 30 cores only local and 1-hop memory is accessed and growth in number of cycles is moderate (under 5%). From 30 to 42 cores, both one-hop and two-hops remote memory are accessed and this results in a more pronounced increase in number of cycles incurred by the program.

For the ARM Cortex-A9, we show in Figure 4.7(g) and (h) both the effects of changing the number of cores and the core frequency. This allows us to validate both the accuracy of the M/M/1 queueing model and of the dependency between core frequency and memory response time, as described by Equation 3.28.

Next we show summary results for all applications. The goodness-of-fit for determining the linearity of $\frac{1}{c(n)}$, as shown in table 4.7 for $n = 1$ to 4 on Intel UMA, $n = 1$ to 12 on Intel NUMA and AMD NUMA, further confirms the accuracy of our model. There is a correlation between the goodness of fit R^2 and the degree of memory contention. Programs *EP.C* and *x264.native* show lower collinearity, because they exhibit the smallest degree of contention. This confirms that the M/M/1 queueing model does not explain their behavior very well, because they

System	Goodness-of-fit, R^2, for Programs								
	EP	***IS***	***FT***	***CG***	***BT***	***SP***	***x264***	***blackscholes***	***memcached***
Intel UMA	0.86	0.97	1.00	0.96	0.99	0.97	0.87	0.89	0.91
Intel NUMA	0.91	0.98	0.99	0.92	0.98	0.96	0.85	0.84	0.82
AMD NUMA	0.90	0.99	1.00	0.99	0.97	0.99	0.81	0.91	0.79

Table 4.7: Goodness-of-fit of CPU cycles model

are bursty. R^2 is close to 1 (i.e. perfect collinearity of $1/c(n)$) when the memory overhead is high. The accuracy of the M/M/1 model for describing the behavior of programs with large memory contention further confirms the non-bursty nature of programs with large memory contention.

Next we present an analysis of the growth of memory contention for $CG.C$. The program exhibits high degree of memory contention of 1.8 to 3.3 times as compared with a sequential execution. On Intel UMA, the contention closely follows how many cores are used in each processor. For one to four cores, the increase in $\omega_\mu(n)$ is due to contention of the shared bus, since all cores within one processor share the same memory bus. From four to five cores, the increase in contention is small, since memory requests by the fifth core, which is allocated in a new processor, uses the bus of the new processor. When the buses and the memory controllers in both processors reach maximum load, contention is most severe as can be seen from increasing the number of cores from seven to eight. On both Intel NUMA and AMD NUMA, the degree of memory contention is smaller than on UMA for similar number of cores. However, the pattern of growth still has a per-processor shape. From one to twelve cores, $\omega_\mu(n)$ increases non-linearly, which shows that the local memory controller of processor one become saturated. When the thirteenth core (located in processor two) is activated, the memory controller of processor two takes over a fraction of the memory requests from processor one controller, reducing the contention. This is why there is a clear decrease in $\omega_\mu(n)$ from twelve to thirteen. There are other reasons that lead to

better NUMA performance compared to the UMA system, such as the larger cache size, faster bus speed and larger memory bandwidth. Overall, the programs that show a larger degree of memory contention on UMA also manifest large contention on NUMA.

Memory contention can be broadly characterized as high, as shown in Figure 4.7(a), (c) and (e) and low, as in Figure 4.7(b), (d) and (f) . However the mapping between problem size and degree of contention is not bijective. Low problem size results in low contention for all programs analyzed by us. This is due to the size of the working set which is of comparable size to the caches of the system. However, for large problem size, there are two cases. In the first case, *EP.C* and *x264.native* have large working set (920 MB for *EP*, 400 MB for *x264*), much larger than the cache of the system, yet do not result in large contention. This is because their pattern of accessing the memory results in low number of cache misses and therefore their performance does not depend significantly on the memory bandwidth. In contrast, the second case, of CG, FT, and SP, their large problem size also translates in large contention. The program with the largest observed contention, the pentadiagonal-solver SP access memories along all dimensions of a multi-dimensional space. Such complex data access patterns leads to large number of cache misses. This results in $SP.C$ having the largest values of contention, with $\omega_\mu(8) = 7.3$ on Intel UMA and $\omega_\mu(24) = 10.5$ on Intel NUMA.

4.4.2 I/O Overhead

We use program *memcached* to validate the I/O overhead model on both traditional and low-power systems. For each system we conduct a set of experiments using the *memslap* client that continuously sends requests to the system under test. The client is running on a separate system, termed *controller systems* (Intel Core-i7), connected to the system under test using a 1 Gbps Ethernet link.

For each experiment, we use memslap to *set* a number of key-value objects into *memcached*, and to *get* a series of these objects. The number of set and get requests, shown in Table 4.8, is chosen such that *memcached* achieves a hit rate close to 100%. For the ARM Cortex-A9 system we have used a smaller cache

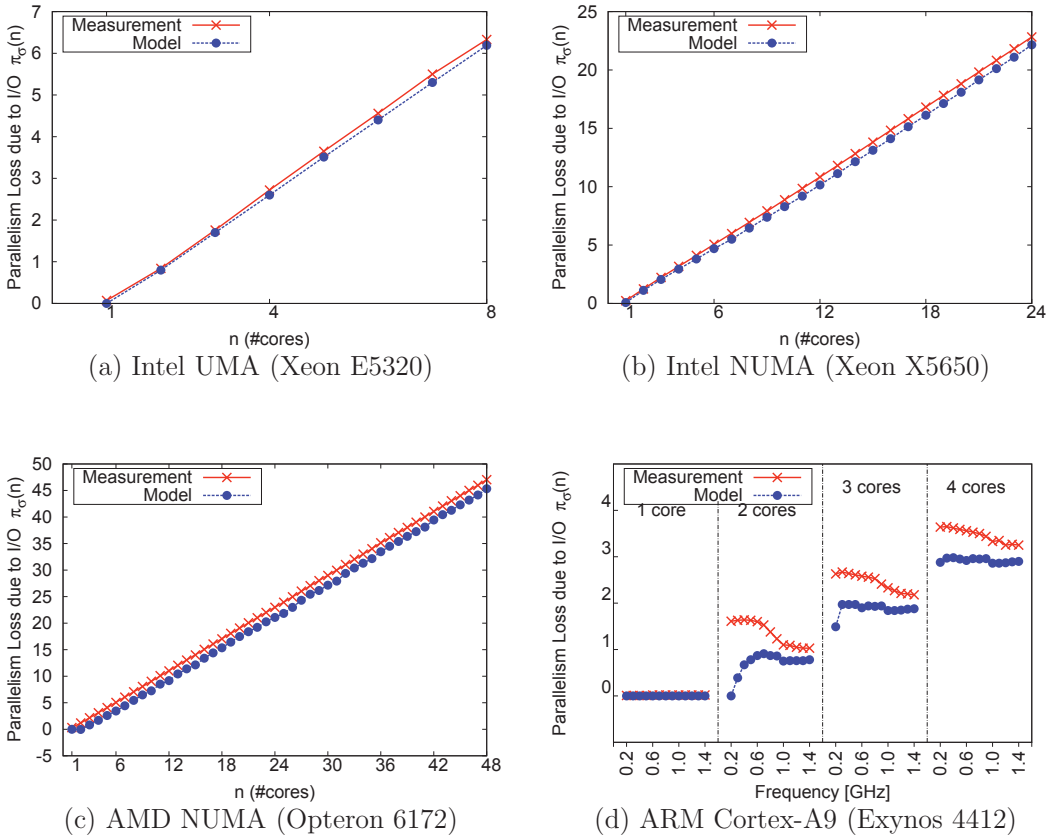
System	Set req.	Get req.	Read bytes [MB]	Written bytes [MB]	Cache size
Intel UMA	200,000	1,800,000	382	1,898	1024 MB
Intel NUMA	200,000	1,800,000	382	1,898	1024 MB
AMD NUMA	200,000	1,800,000	382	1,898	1024 MB
ARM Cortex-A9	60,000	540,000	115	570	256 MB

Table 4.8: Workload parameters for *memcached*

size and a smaller number of requests, to achieve a similar execution time and hit rate with the traditional x64 multicore systems. The get requests are performed in batches of 12 operations per one request. The requests are completely independent and can be serviced in parallel by *memcached*, thus the parallelism loss due to data dependency, $\pi_d \approx 0$. Therefore, any parallelism loss is caused only by memory contention and by I/O overhead.

Next we show the comparison of the modeled parallelism loss due to I/O overhead, ω_σ, with the measured values. The measured values of ω_σ are obtained by measuring the average number of cores that are not doing neither useful work, nor active due to memory contention, using the `perf` performance monitoring tool.

Figure 4.8 shows the comparison between modeled and the measured parallelism loss due to I/O overhead. There are two trends that are consistent across all four validation experiments. First, the parallelism loss due to I/O overhead, π_σ grows almost linearly with n. The reason is that for all the multicore systems, the network I/O is almost always the system bottleneck. For traditional multicores, an execution on one core completely saturates the I/O bandwidth. Thus, using more than one core when running *memcached* does not lead to improvements in execution time: all the hardware parallelism is lost due to waiting on the network I/O device. For the low-power ARM Cortex-A9 system, we also show how the



(a) Intel UMA (Xeon E5320)
(b) Intel NUMA (Xeon X5650)
(c) AMD NUMA (Opteron 6172)
(d) ARM Cortex-A9 (Exynos 4412)

Figure 4.8: Validation of the I/O overhead model: *memcached*

parallelism loss depends also on the core frequency f. For small core frequencies the CPU time W dominates the execution time, and *memcached* is CPU-bounded. But at the CPU frequency approaches 1.4 GHz, W decreases, and the imbalance between CPU and I/O reduces. However, using one core even at maximum frequency does not fully saturate the memory bandwidth. The knee-point where the system transitions from CPU-bounded to I/O-bounded execution is $n = 2$ cores and frequency $f = 0.6GHz$. Beyond this configuration point, increasing the core frequency or count does not reduce execution time, and the parallelism loss due to I/O overhead starts to increase.

The second observation is that the model has good accuracy, but tends to underestimate the parallelism loss due to I/O overhead. The model inaccuracy for the traditional multicore system is within 3% (Intel NUMA) and 5% (Intel UMA)

when comparing the parallelism loss directly, and less than 15% when comparing the modeled I/O throughput with the measured throughput. However, for the low-power multicore, the accuracy is significantly worse, with an average error of 22%.

The larger modeling error on the low-power multicore is caused by the assumption that the total work performed by *memcached* remains constant when the number of cores and core frequency changes. Even though we fixed the problem size across experiments, the total number of instructions and work cycles does not remain constant when the number of cores and clock frequency changes. There are two reasons for this: (i) *Memcached* uses a polling mechanism to check if the network sockets have any available data; this mechanism incurs more instructions as the number of threads is increased. (ii) The low-power ARM Cortex-A9 executes the code of the network interrupts and of the network driver only the first core. Thus, even if we assume that *memcached* is perfectly parallelizable, in practice all the kernel-level code is executed on a single core. This is particularly worse on executions on low core frequencies, when the first core uses a disproportionately large number of cycles to service the network interrupts and the TCP/IP code. Our model assumes all these cycles can be fully parallelizable among cores, hence underestimating the parallelism loss due to I/O overhead.

4.4.3 Exploited Parallelism

To validate the exploited parallelism prediction we compare modeled values of exploited parallelism against speedup measurements. To evaluate the accuracy of the data dependency model independent of the accuracy of the memory contention model, we use predicted and measured values of $\omega_\mu(n)$ in modeling $\pi'(m, n = m)$, and present both. Firstly, we validate the speedup of programs partitioned in different number of threads, and running on enough cores to execute them

concurrently. Secondly, we fix the number of threads and execute them on different number of cores.

Program	**m**	**Measured** $\pi'(\mathbf{m}, \mathbf{n} = \mathbf{m})$	**Modeled** $\pi'(\mathbf{m}, \mathbf{n} = \mathbf{m})$	
			measured ω_μ	**modeled** ω_μ
BT.C	2	1.77	1.80	1.80
	4	2.52	2.65	2.52
	8	3.50	3.91	3.50
EP.C	2	1.98	1.99	1.99
	4	3.96	3.99	3.99
	8	7.83	7.98	7.98
FT.B	2	1.63	1.72	1.72
	4	2.23	2.33	2.30
	8	2.80	2.83	3.11
IS.C	2	1.93	1.95	1.95
	4	3.45	3.47	3.69
	8	5.02	5.08	6.65
CG.C	2	1.75	1.82	1.82
	4	2.27	2.20	2.76
	8	2.33	4.18	3.91
SP.C	2	1.32	1.30	1.30
	4	0.99	0.95	0.86
	8	0.97	0.81	0.83
x264.native	2	1.91	1.88	1.91
	4	3.67	3.47	3.60
	8	4.91	4.70	4.65
blackscholes.native	2	1.82	1.73	1.73
	4	3.06	3.30	3.20
	8	3.71	4.18	4.11
memcached.memslap	2	0.95	0.99	0.99
	4	1.07	1.05	1.02
	8	0.96	1.01	1.00

Table 4.9: Model vs measured exploited parallelism on Intel UMA

For Intel UMA, we partition all six program into 2, 4 and 8 threads and perform baseline runs on one core to derive the average number of active threads. Using both modeled and measured values of $\omega_\mu(n)$, we compare the speedup model against measurements, when the program is running on 2, 4 and 8 cores. Table 4.9 shows the validation results. The average relative error of the model is 7.5% for

measured ω_μ and 11.3% for modeled ω_μ. In table 4.10 we show the validation results for system Intel NUMA. The dwarfs are partitioned into 2, 4, 8, 12, 16 and 24 threads and validated against speedup measurements performed on a number of cores equal with number of threads. For Intel NUMA, all cores are allocated on the first socket for the runs with 2, 4, 8 and 12 threads, but the cores are divided equally between the sockets for runs with 16 and 24 threads. For AMD NUMA we used 4, 8, 12, 24 and 48 cores, using a fill-socket-first policy, and the validation results are shown in table 4.11.

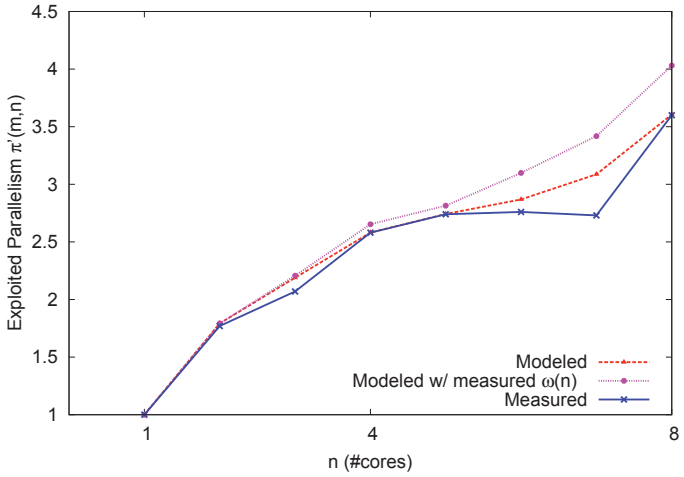


Figure 4.9: Modeled vs measured exploited parallelism: *BT.C* on Intel UMA

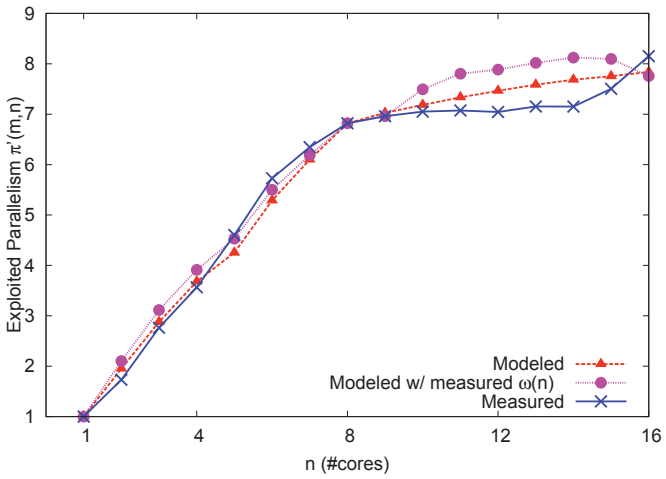


Figure 4.10: Modeled vs measured exploited parallelism: *BT.C* on Intel NUMA

Program	**m**	**Measured** $\pi'(\mathbf{m}, \mathbf{n} = \mathbf{m})$	**Modeled** $\pi'(\mathbf{m}, \mathbf{n} = \mathbf{m})$	
			measured ω_μ	**modeled** ω_μ
BT.C	2	1.88	1.82	1.80
	4	2.62	2.64	2.58
	8	3.80	3.91	3.69
	12	3.92	3.91	3.60
	24	5.02	4.98	4.90
EP.C	2	1.98	1.99	1.99
	4	3.96	3.99	3.99
	8	7.83	7.98	7.98
	12	11.53	11.81	11.90
	24	20.52	20.81	20.92
FT.C	2	1.83	1.68	1.78
	4	2.73	2.70	2.70
	8	3.11	3.00	3.08
	12	4.52	4.48	4.50
	24	7.85	7.91	7.70
IS.C	2	1.93	1.95	1.95
	4	3.95	3.87	3.90
	8	5.73	5.80	5.80
	12	9.52	9.78	9.65
	24	14.15	14.54	14.92
CG.C	2	1.88	1.88	1.84
	4	2.35	2.39	2.96
	8	3.87	3.65	3.18
	12	4.58	4.41	4.50
	24	5.56	5.50	5.38
SP.C	2	1.42	1.40	1.40
	4	1.49	1.59	1.56
	8	1.51	1.58	1.53
	12	1.48	1.72	1.52
	24	1.90	1.91	1.81
x264.native	2	1.78	1.60	1.62
	4	3.45	3.40	3.43
	8	6.47	6.21	6.32
	12	7.70	7.52	7.60
	24	9.76	9.40	9.42
blackscholes.native	2	1.73	1.72	1.71
	4	2.90	2.71	2.70
	8	4.24	4.20	4.17
	12	5.04	5.10	5.07
	24	5.65	5.50	5.44
memcached.memslap	2	0.98	0.98	0.98
	4	0.96	0.96	0.96
	8	1.01	1.00	1.00
	12	1.01	1.01	1.01
	24	1.01	1.00	1.00

Table 4.10: Model vs measured exploited parallelism on Intel NUMA

Program	**m**	**Measured** $\pi'(\mathbf{m}, \mathbf{n} = \mathbf{m})$	**Modeled** $\pi'(\mathbf{m}, \mathbf{n} = \mathbf{m})$	
			measured ω_μ	**modeled** ω_μ
BT.C	4	3.63	3.56	3.73
	8	6.47	6.43	6.76
	12	9.39	9.14	9.14
	24	17.43	16.01	14.35
	48	23.40	21.11	20.94
EP.C	4	4.00	3.99	3.99
	8	7.79	7.90	7.87
	12	11.95	11.64	11.64
	24	22.23	21.97	22.30
	48	33.44	33.01	45.91
FT.C	4	3.41	3.41	3.69
	8	6.03	6.13	6.58
	12	8.64	8.68	8.68
	24	13.46	14.21	13.86
	48	21.66	26.74	22.87
IS.C	4	3.95	4.24	4.24
	8	7.30	8.12	7.98
	12	10.23	11.24	11.24
	24	15.60	16.72	17.69
	48	26.36	26.69	25.26
CG.C	4	3.85	4.00	3.59
	8	5.90	6.86	5.83
	12	7.20	6.89	6.89
	24	12.20	12.28	10.56
	48	15.77	16.63	17.55
SP.C	4	2.22	2.19	3.24
	8	3.09	3.41	4.55
	12	3.86	4.07	4.07
	24	6.81	7.08	4.85
	48	7.61	8.66	6.47
x264.native	4	3.55	3.51	3.50
	8	6.56	6.50	6.47
	12	9.21	8.98	8.90
	24	13.29	13.11	13.00
	48	14.29	13.14	13.10
blackscholes.native	4	2.92	3.11	3.00
	8	4.36	4.50	4.47
	12	5.15	5.48	5.40
	24	6.32	6.41	6.36
	48	6.60	6.86	6.50
memcached.memslap	4	0.96	0.96	0.96
	8	1.01	1.00	1.00
	12	1.00	1.01	1.01
	24	1.00	1.01	1.00
	48	1.03	1.03	1.03

Table 4.11: Model vs measured exploited parallelism on AMD NUMA

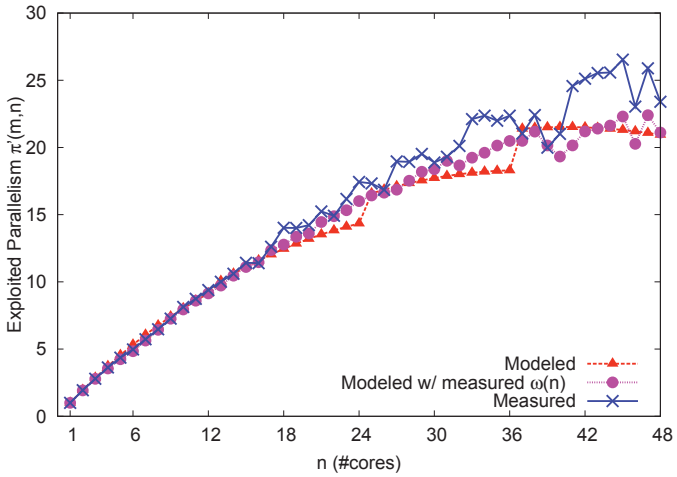


Figure 4.11: Modeled vs measured exploited parallelism: $BT.C$ on AMD NUMA

We partition BT.C in m=8 threads for UMA system, m=16 threads on Intel NUMA and $m = 48$ on AMD NUMA. We determine $\pi(m, \infty)$ using baseline runs on one core. For the $\pi'(m, n)$ prediction, we used values of $\omega_\mu(n)$ predicted by our model and values directly measured, and we present both. We compared our prediction against speedup measurements, keeping the number of threads fixed and increasing the number of cores n from 1 to m. For the Intel NUMA system, the increase in the number of cores is done as follows:

- $n \in [1 : 4]$ activate the first hardware thread of the cores from the first socket;
- $n \in [5 : 8]$ activate the first hardware thread of the cores from the second socket;
- $n \in [9 : 12]$ activate the second hardware thread of the cores from the first socket;
- $n \in [12 : 16]$ activate the second hardware thead of the cores from the second socket.

On the AMD NUMA system, the order of cores activation follows a fill-socket-first

policy, and the NUMA nodes are activated in increasing latency order, starting from NUMA node 0.

Figures 4.9, 4.10 and 4.11 show the predicted versus measured exploited parallelism on Intel UMA, Intel NUMA and AMD NUMA systems. The evaluate the accuracy of the data-dependency model, we predict the exploited parallelism using both predicted and modeled values of memory contention factor ω_μ. The control metric is the measured speedup.

For BT.C we noticed that the measured speedup varies among the five runs, especially when the number of threads m does not divide by the number of cores n. Considering that measured speedup is our control value, the accuracy of the model is the lowest when m does not divide by n (maximum error is 25% for predicted $\pi'(8,7)$ using measured ω_μ and 13% for predicted $\pi'(8,7)$ using modeled ω_μ). For the other cases, the model has much better accuracy. Overall, the average error for BT.C on UMA is 9% for predicted $\pi'(m,n)$ using measured $\omega_\mu(n)$ and 4% for predicted $\pi'(m,n)$ using modeled $\omega_\mu(n)$.

4.4.4 Power and Energy

We validate the energy model against measurements of energy usage, as described in section 4.1. First we show summary validation results for all workloads. We present in detail the validation of programs *EP*, *CG*, *SP* and *memcached*, and present summary results for the rest of the programs.

To validate the programs, we apply the baseline run for each program. The baseline runs are performed on four core-frequency configurations: $n = 1$ $f =$ 0.2 GHz, $n = 1$ $f = 1.4$ GHz and $n = 4$ $f = 0.2$ GHz, $n = 4$ $f = 1.4$ GHz. On each configuration we measure the total cycles, stall cycles, number of I/O requests, arrival rate of I/O requests. To validate the model, we predict the execution time $T(n,f)$ and energy $E(n,f)$ considering the number of cores n

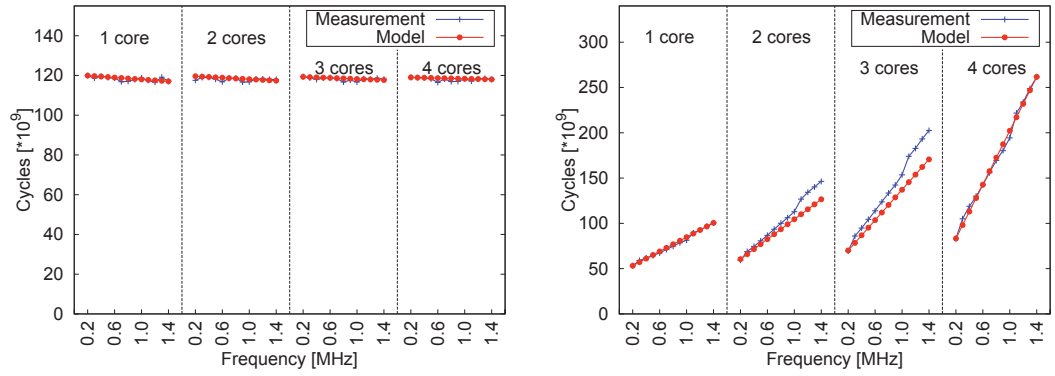
and clock frequency f fixed throughout the program execution. We compare this prediction against measurements of $T(n, f)$ and $E(n, f)$. To change the number of cores that the program is using we change the number of worker threads of the programs. For each program, we validate four cores and 13 core frequencies, giving 52 core-frequency configurations.

We measure the energy usage of the program using a Yokogawa WT210 power monitor. The power monitor measures the DC voltage supply, the DC current, the power and the energy usage since a fixed arbitrary time moment (usually since the last manual reset of the power monitor). Because the sample size is quite coarse, we use the measured value of energy as the control metric for the power and energy models. Thus, in the experiments, we log the energy usage at the beginning and the end of the experiment. The measured total energy usage is then computed as the difference between the two values.

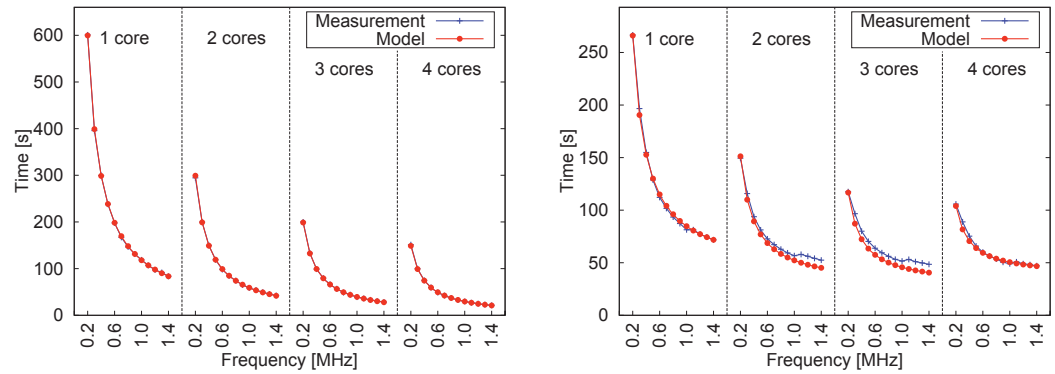
To evaluate the accuracy of the energy model independent of the accuracy of the prediction of the CPU time and I/O time prediction, we present the validation for both execution time and energy usage. For execution time validation we also present the validation of the predicted number of cycles, to further isolate the source of inaccuracy. Similarly, the validation of the energy model is broken down into validation of energy usage and of power use.

Figures 4.12 show the validation of the embarrassingly-parallel CPU-intensive *EP* program, and the validation of a program with a medium degree of memory contention, *CG*. The accuracy of the parallelism model is very good, with prediction error rate of less than 1% percent for *EP* and less than 6% for *CG*. However, the accuracy of the energy model is slightly worse: *EP* is 22% and *CG* is 11%.

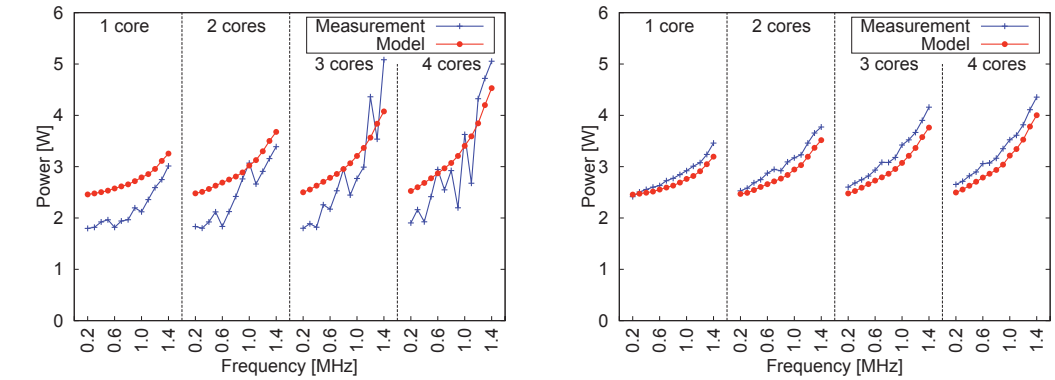
Figures 4.13 present the validation results for the most memory-bounded program, *SP* and for I/O-bounded *memcached*. The accuracy of the programs follows the same pattern as for *EP* and *CG*: the power model is the main culprit for the



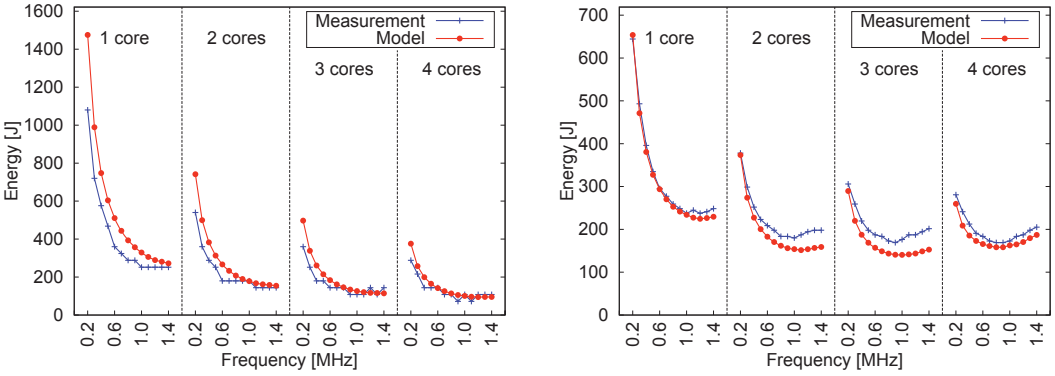
(a) EP (b) CG



(c) EP (d) CG

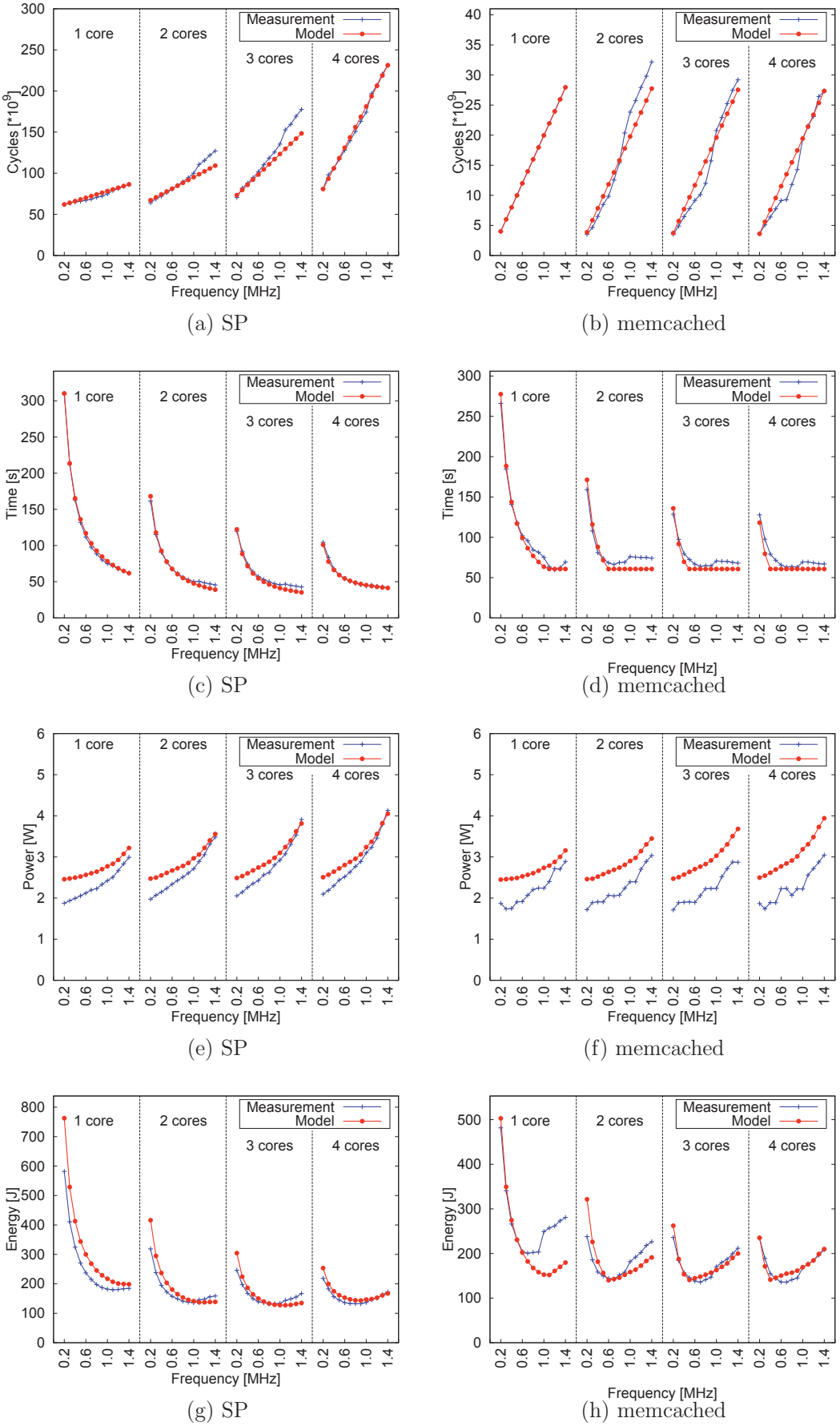


(e) EP (f) CG



(g) EP (h) CG

Figure 4.12: Validation of power-energy model: *EP* and *CG*



(a) SP (b) memcached

(c) SP (d) memcached

(e) SP (f) memcached

(g) SP (h) memcached

Figure 4.13: Validation of power-energy model: *SP* and *memcached*

loss of accuracy of the energy model, with the execution time being much more accurate than the power model.

4.4.5 Errors and Limitations

We have presented the validation results for the memory contention, I/O overhead, exploited parallelism and power/energy models. Next we present the overall accuracy of our modeling approaches, and discuss the model errors and limitations.

Table 4.12 shows the average error between model and measured values, using the six HPC dwarfs, two PARSEC programs and the *memcached* workload. We present error for all the four proposed models for memory contention factor ω_μ, I/O overhead factor ω_σ, exploited parallelism $\pi'(m,n)$ and energy usage of the program $E(n)$. The error is determined across predictions of the model for all values of n, while for ARM Cortex-A9, for all values of n and f.

The average error between a set of predicted values $x_1, x_2 \ldots x_n$ and the set of measured value $y_1, y_2, \ldots y_n$ is computed as:

$$Err[\%] = \frac{100}{n} \sum_{i=1}^{n} \frac{abs(y_1 - x_1)}{y_1} \tag{4.1}$$

Two factors affect the accuracy of the exploited parallelism model: (i) the inaccuracy of $\omega_\mu(n)$, which is the most significant source of error for the HPC dwarfs and (ii) inaccuracy of $\pi(m,n)$. We hypothesize that $\pi(m,n)$ is related to inter-barrier time of the program. We observe a variation of up to 11% for *IS.C* and 23% for *CG.C* among the values of the average number of active threads predicted from the five baseline runs. We suspect that the inter-barrier time both programs may be smaller than our sample size of 10 ms, which is in-line with observations made by [55]. Overall, the average error across all programs is around 6% for UMA and 11% for NUMA. *EP* is the most straightforward to model, with an

System	Program	Error [%]			
		ω_μ	ω_σ	$\pi'(n)$	$E(n)$
Intel UMA	*EP*	0.35	–	1.22	–
	IS	3.50	–	11.55	–
	CG	9.70	–	12.20	–
	FT	3.37	–	9.11	–
	BT	2.62	–	5.39	–
	SP	9.75	–	9.80	–
	x264	8.18	–	8.50	–
	blackscholes	9.90	–	3.31	–
	memcached	9.07	4.80	6.03	–
Average		6.27	4.80	7.45	–
Intel NUMA	*EP*	8.70	–	6.11	–
	IS	7.14	–	4.10	–
	CG	10.27	–	12.03	–
	FT	14.91	–	16.22	–
	BT	5.55	–	10.21	–
	SP	24.44	–	28.30	–
	x264	9.19	–	9.91	–
	blackscholes	11.11	–	13.30	–
	memcached	3.20	15.84	3.00	–
Average		10.50	15.84	11.48	–
AMD NUMA	*EP*	1.83	–	11.31	–
	IS	2.53	–	9.10	–
	CG	9.69	–	16.65	–
	FT	6.18	–	14.12	–
	BT	4.13	–	15.80	–
	SP	14.25	–	21.50	–
	x264	1.91	–	14.23	–
	blackscholes	3.20	–	6.55	–
	memcached	1.05	3.92	2.0	–
Average		4.97	3.92	12.56	–
ARM Cortex-A9	*EP*	0.59	–	1.63	22.38
	IS	2.72	–	2.91	7.67
	CG	5.59	–	17.06	11.07
	FT	4.19	–	3.40	8.63
	BT	3.76	–	3.60	7.81
	SP	4.56	–	4.85	13.23
	x264	0.91	–	3.88	12.55
	blackscholes	1.20	–	4.54	9.91
	memcached	9.40	22.91	9.40	10.70
Average		3.65	22.91	5.70	11.55

Table 4.12: Model errors

average error of less than 1%. Programs *x264*, *blackscholes* and *memcached* also show good accuracy (generally less than 10% error). Overall, the inaccuracy in the prediction of $\pi(m, n)$ results in only a small prediction error. These results show that the dynamic run-queue size is a good proxy for determining the average number of active threads of a program.

However, a larger source of inaccuracy in the parallelism model on the benchmarked programs is the prediction of the memory contention factor, ω_μ. When using one or two sockets, the model accuracy is generally good and the variability among runs is limited. However, on the AMD NUMA system, using more than two NUMA domains results in a noticeable decrease in model accuracy. We identify two factors for this loss in accuracy:

1. The assumption of equal memory affinity among threads is violated for complex NUMA topologies. In all the programs covered by our validation, the division of work among threads is equal at an algorithmic level, and thus the number of memory accesses to shared variables should be equal among threads. However, the threads almost inevitably exhibit imbalances during their executions, and the imbalances lead to a skew in memory access patterns. This observation is consistent to findings reported in the literature [79].

2. Executions on larger number of cores on AMD NUMA exhibit larger variability in the number of cycles incurred on the same configuration. This variability cannot be controlled by repeating the experiment several times and averaging the results, as the system exhibits clustering of performance results. For example, the run of $BT.C$ on 42 cores registers is executed ten times. The first three runs register an execution time of 9.58 seconds and a standard deviation of 0.04 seconds. The fourth and fifth run register 10.25

seconds and 10.03 seconds, respectively. The next four runs register an average of 7.04 and a standard deviation of 0.04 seconds, while the tenth run executes in 7.22 seconds. This shows that the execution times are clustered around at least two centers (with 7.04 and 9.58 seconds) but with significant points far from both centers. We briefly investigated the reason for the large variations among runs, by periodically running the `numactl --hardware` command to check the free space on each NUMA node. To our surprise, we noticed huge variations among the memory used in different NUMA nodes, even if all our workloads should have uniform memory affinity. Many runs allocate a disproportionate amount of memory on the first NUMA node. For example, on *BT* using 42 cores, a balanced memory allocation should allocate 28% of the working set size on each of the first three NUMA nodes and 14% on the fourth. However, we routinely observe more than 50% of the working set size allocated on the first NUMA node. However, we cannot balance this memory allocation just by tweaking the NUMA policy. The OS NUMA policy only dictates *which* nodes can be used during the memory allocation. It does not specify *how much* of the working set size should be allocated on each node. Similar observations on the suboptimal behavior of the NPB HPC dwarfs on complex NUMA topologies are presented by [109].

We conclude that our simple M/M/1-based approach for predicting the performance of memory-bounded applications is reasonable for simple NUMA topologies, but better support for understanding and controlling the way the OS handles the application of the NUMA policies is needed both for performance prediction and for performance optimizations.

For programs with I/O overheads, we identify three factors that affect the accuracy of the model. The most significant source of error comes from irregularities during execution. For example, *memcached* incurs more instructions on

higher core frequencies, which are caused by a polling mechanism used to monitor the network sockets. This significantly increases the energy used, but does not reduce the execution time. This increase causes our model to underestimate by up to 23% the CPU cycles incurred by *memcached* on one core.

In the power and energy models, there are three main sources of inaccuracy:

1. The first source of errors is the accuracy of the system characterization parameters. In particular, the power values for active cycles, stall cycles and idleness differ by up to 20 mW. When no other significant sources of error are present, this variability translates into a slight overestimate of the average power, especially for configurations with low frequencies or low core counts.

2. The second source of error is the low accuracy of the measured energy. The resolution of our energy meter is 0.001 Wh or 3.6 J. Because the measuring sample size is 1 second, a program consuming less than 3.6 W might register two consecutive energy samples with the same energy value.

3. The third error is the instability of the measured power values. The system exhibits large variations in power usage even for seemingly the same workload executed in the same configurations. One tentative explanation is that the power consumption is linked to the temperature of the Exynos 4412 SoC. Although the system is provisioned with a heat-sink that can passively dissipate enough heat to keep the SoC well below the maximum operating temperature, long running CPU-intensive jobs lead to an increase the temperature of the heat-sink to around 45 °C. We did not possess the tools to do a systematic analysis of the dependency between temperature and power usage, but we noticed an idle system power of more than 1850 mW immediately after running a job on a high temperature SoC. After the

job finishes, the idle power slowly and consistently drops to a minimum of 1740 mW, which corresponds to the minimum observed temperature of the heat-sink of 21 °C.

Next we discuss the main limitations of our modeling approach. The main model limitation is the requirement that the three types of service demands – to cores, memory and I/O devices – are perfectly overlapped. While this assumption is reasonable for HPC dwarfs, the analyzed PARSEC programs and server workloads such as *memcached*, not all programs can be faithfully modeled as such. In particular, a class of programs which does not obey this overlap very well consists of programs with a long setup phase during which the data is loaded from the storage, followed by a compute phase [111]. If this setup phase cannot be overlapped with the beginning of the computations, our model might mispredict the performance of such programs. This limitation can be overcome if the model is first applied to the setup phase and then for the computation phase. Programs *x264* and *blackscholes* both exhibit a setup phase during which they load a file from disk. While *x264* loads pieces of a file periodically, overlapping the reading from disk with computations, *blackscholes* first loads the entire file in memory. Due to this aspect, *blackscholes* would be more accurately predicted as a program with two distinct phases. But overall, the loading phase is small enough such that the computation phase dominates. Furthermore, it is beyond the objective of this thesis to tailor the model for program-specific behaviors.

A second model limitation is that it assumes that programs obey a work-conserving property among different core-frequency configurations. If a program executes in one way on one core, but in a different way when the number of cores changes, it is not work-conserving because the total work performed by the program changes among configurations. Because we base our predictions on measurements of the total number of work cycles on one core, we might mispredict the

useful work of the program. Unfortunately, complex programs rarely have a perfect work-conserving property. Even among our analyzed programs, *memcached* and even *EP* exhibit changes among the number of instructions executed across different core-configurations. While for *EP* the change is negligible and mostly observable on NUMA systems, for *memcached* it is noticeable on the low-power ARM Cortex-A9 system, where the main source of inaccuracy is linked to the changes in number of instructions among different core-configurations, even when the input of the program is the same.

4.5 Summary

This section describes four experimental evaluations. First we perform an sensitivity analysis that helps us decide the parameters of the baseline executions used to collect the inputs of our model. In the second analysis, we present results that support our modeling assumption that the number of active core does not influence strongly the number of work cycles, CPU instructions and last-level cache misses. Third, we showed that the memory traffic is not always bursty, as previously reported in literature. Instead, the memory burstiness depends on the problem size. Large parallel programs that exhibit memory contention generate non-bursty memory traffic, while programs with low contention do generate bursty traffic. Lastly, validation results of our model against measurements show an average error across all programs and problem sizes of around 7% for Intel UMA, 11% for Intel NUMA, 13% for AMD NUMA, and 6%-12% for ARM Cortex-A9.

Chapter 5

Model Applications

In this section we describe applications for our parallelism-energy modeling approach for understanding and optimizing the performance of shared-memory programs. First we present a summary of the performance of all workloads studied in this thesis and discuss the causes of the performance loss in each program. Next, we use our proposed models to address three scenarios that may be encountered by users and architects of parallel systems:

1. Given a program with a specified problem size and a target multicore system, what is the configuration with the minimum number of cores, required to meet a specified execution time deadline T_D? If it cannot be achieved it, where does the program lose parallelism? And how can the system and the program be improved to meet the performance requirement?

2. Given a program with a specified problem size and a target multicore system, what is the configuration that achieves minimum energy usage? How does this configuration depend on the type of system bottleneck? Can we optimize the time and energy usage compared to the default Linux policies of setting the number of cores and core clock frequency?

3. Given a program executing on a multicore system with an imbalance between cores, memory and I/O, what is the best strategy for reducing the energy wastage: lower the hardware performance to reduce power, or improve the hardware performance to reduce execution time?

5.1 Understanding Parallelism-Energy Performance

We present a summary of parallelism and energy performance for the studied programs, and discuss the causes of the parallelism loss for the programs.

5.1.1 Inherent and Exploited Parallelism

Table 5.1 shows the inherent parallelism, exploited parallelism and parallelism losses due to data dependency, memory contention and I/O overhead of the profiled applications on the x64 multicore systems. Each application is partitioned in a number of threads m equal to the number of cores of the machine. For each application, the most significant parallelism loss is highlighted in bold.

The performance summary shows that the programs chosen by us exhibit widely different inherent and exploited parallelism. HPC programs typically have high inherent parallelism, as there is little data dependency between the threads of the programs. This matches the analysis done by previous work [40] and is linked to the application structure of the HPC dwarfs. All HPC programs are OpenMP programs written in C or Fortran that have long parallel regions that finish without an OpenMP barrier (because they use the OpenMP `nowait` attribute). Thus, the small loss due to data dependency is mostly due to initialization and cleanup and due to imperfect load balancing of the OpenMP for loop

System	Program	Parallelism		Parallelism Loss		
		Inherent	Exploited	Data Dependency	Memory Contention	I/O Overhead
		π	π'	π_δ	π_μ	π_σ
Intel UMA ($m = 8$)	*EP*	7.99	7.83	0.01	0.16	–
	IS	7.80	6.65	0.20	**1.15**	–
	CG	7.70	3.91	0.30	**3.79**	–
	FT	7.67	3.11	0.33	**4.56**	–
	BT	7.33	3.50	0.67	**3.83**	–
	SP	7.85	0.83	0.05	**7.12**	–
	x264	6.03	5.65	**1.97**	0.38	–
	blackscholes	6.12	5.91	**1.88**	0.21	–
	memcached	8.00	1.00	0.00	0.65	**6.35**
Intel NUMA ($m = 24$)	*EP*	23.57	20.92	0.43	2.65	–
	IS	21.80	14.92	2.20	**6.88**	–
	CG	23.15	5.38	0.85	**17.77**	–
	FT	22.77	7.70	1.23	**16.07**	–
	BT	23.18	4.90	0.82	**18.28**	–
	SP	22.90	1.81	1.10	**21.09**	–
	x264	11.23	9.92	**12.77**	1.31	–
	blackscholes	7.87	7.12	**16.13**	0.75	–
	memcached	24.00	1.00	0.00	0.44	**22.56**
AMD NUMA ($m = 48$)	*EP*	47.03	45.11	0.97	1.92	–
	IS	46.15	25.25	1.85	**20.90**	–
	CG	45.80	17.55	2.20	**28.25**	–
	FT	44.38	22.87	3.62	**21.51**	–
	BT	46.75	20.94	1.25	**25.81**	–
	SP	47.21	6.47	0.79	**40.74**	–
	x264	16.24	14.29	**31.76**	1.95	–
	blackscholes	6.85	6.50	**41.15**	0.35	–
	memcached	48.00	1.03	0.00	0.74	**46.23**

Table 5.1: Inherent parallelism, exploited parallelism and parallelism loss

iterations to threads, which causes some threads to finish the last for loop earlier. *Memcached* also exhibits very low data dependency, as it uses a multi-threaded mechanism for listening on sockets, using the `epoll_wait` system call. For the time interval when the *memcached* performs service, each request is assigned to a different thread. Because there are much more concurrent requests than threads, and because the I/O bandwidth is much smaller than what the threads service capacity, the threads only wait on the network sockets (distinguished by being blocked on `epoll_wait`, and not on locks or semaphores, when they are blocked on `futex_wait`). Finally, *memcached* does have startup and a cleanup phases that are not fully parallelizable, but they are not included in our analysis.

The Parsec programs exhibit much larger parallelism loss due to data dependency, mostly due their large initialization phase. For example, *blackscholes* has a startup phase during which it reads an input file into memory and performs a data transformation on the input. This phase is completely sequential, thus resulting in a large parallelism loss due to data-dependency.

The exploited parallelism of HPC dwarfs is limited the most by memory contention. With the exception of EP, all the other dwarfs exhibit non-negligible parallelism loss due to memory contention: high for SP, medium for CG, BT and FT, and low for IS. Among CG, BT and FT, we observe that FT has the highest parallelism loss on Intel UMA, but the lowest on the NUMA systems. From [79] we observe that FT has smaller bandwidth requirements than CG and BT, but more than 50% of memory requests are traversing the NUMA interconnect, compared to less than 20% for CG and BT.

5.1.2 Power and Energy Performance

We discuss the power and energy incurred on the ARM Cortex-A9 system and the energy efficiency factor for the cores, memory and I/O of each program. Ta-

System	Program	Time [s]	Energy [J]	Power [W]	η_{CPU} [%]	η_M [%]	$\eta_{I/O}$ [%]
ARM Cortex-A9	*EP*	21.08	95.5	4.52	99	0	0
	IS	20.33	83.63	4.10	99	99	0
	CG	47.11	18.70	4.00	98	98	0
	FT	14.77	60.23	4.07	99	99	0
	BT	43.50	183.10	4.20	99	99	0
	SP	41.30	167.28	4.04	100	100	0
	x264	692.04	3052.92	4.41	92	0	0
	blackscholes	295.04	1129.33	3.82	82	0	0
	memcached	60.63	209.95	3.46	52	93	100

Table 5.2: Time and energy performance on ARM Cortex-A9 system

ble 5.2 shows the time-energy performance of the programs when using all cores at maximum frequency. For the HPC dwarfs the results are intuitive and tally directly with the know characteristics of the programs. PARSEC programs $x264$ and *blackscholes* have lower CPU energy proportionality, because they lose parallelism due to data-dependency and thus do not keep the cores fully occupied. *Memcached* has even lower energy proportionality because of its parallelism loss due to I/O overhead.

We show in detail the energy proportionality of all cores, memory and I/O device for *memcached* on all cores and clock frequencies, in Figures 5.1 and 5.2. Execution on a small number of cores or small clock frequencies have higher energy

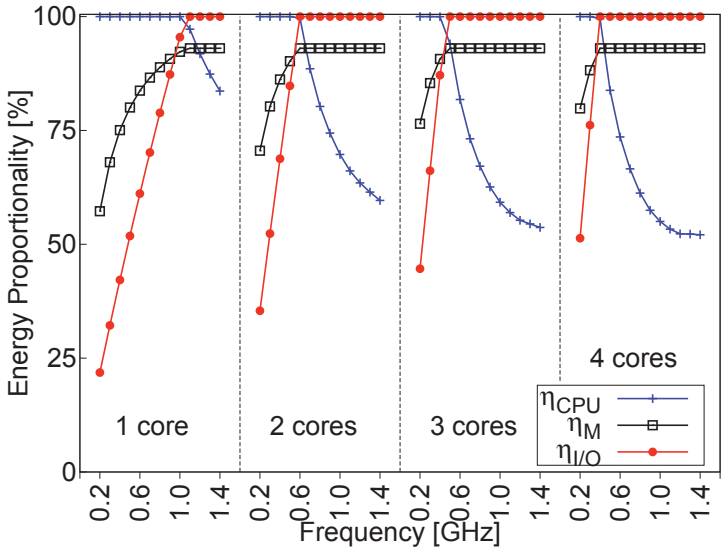


Figure 5.1: CPU, memory and I/O energy proportionality: *memcached*

proportionality than execution on larger number of cores. But this execution incurs both higher execution time and higher energy usage. For example, the execution on one core at minimum frequency incurs $T = 277$ s and $E = 503$ J, while an execution on maximum cores and maximum frequency incurs $T = 60.63$ s and $E = 210$ J.

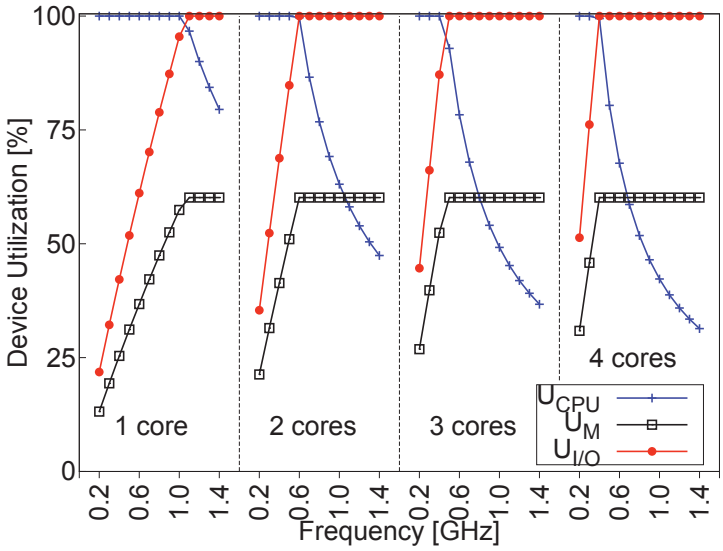


Figure 5.2: CPU, memory and I/O device utilization: *memcached*

Our analysis shows that high energy proportionality does not necessarily mean good performance. For example, with the exception of *EP*, the HPC dwarfs have nearly perfect CPU and memory energy proportionality, because both devices have near full utilization. However, due to memory contention, the CPU does not perform useful work, but rather incurs stall cycles. For the same reason the average power consumption is not as high as in CPU-bounded programs, because CPU power consumption when incurring stall cycles is smaller than when incurring work cycles. This analysis shows that energy proportionality alone is not always a good indicator of the time-energy performance of a program. Throughout this thesis we have used the definition of energy proportionality from literature [16]. However, just relating performance to device utilization does not always paint a clear picture about the time-energy performance of a program. Ideally, the energy

proportionality should also be related to the *useful work* output of the program. In the next two sections we show that our proposed parallelism performance model can be used as a guideline for minimizing both execution time and energy usage, even for programs where the energy proportionality is close to 100%.

5.2 Meeting Performance Requirements on Multicore Systems

Given a large parallel application and a problem size, a common scenario encountered by users is to execute the program faster than a specified deadline T_D. We show an application of our model for predicting the number of cores that achieves this performance, for a shared-memory program with significant data-dependency and memory traffic. Specifically, we determine the multicore configuration with the minimum number of cores n on which a program meets the performance deadline T_D.

5.2.1 Number of Cores Required to Meet a Deadline

Given a program and a problem size, we describe the application of the model to predict the number of cores required to execute the program in less than T_D. We select a target multicore systems with two quad-core sockets connected using a UMA memory architecture. On this target system, our goal is to meet a deadline of execution time $T_D < 100$ seconds for a complex parallel application $SP.B$ (SP with problem size B) [13]. Because SP is an application that allows changing the number of threads in which the program is partitioned, first we chose the partition size. Because the system has eight cores, we choose $m = 8$ threads to use in program SP.

We perform a baseline execution using one core on the target multicore system. During this execution, we collect the trace of operating system run-queue and measure the execution time on one core, $T(1)$, the number of work cycles w and total cycles $c(1)$. Next we perform a baseline execution using two cores on the same socket, and collect $c(2)$. Thirdly, we perform an execution using all four of the first socket and only one core on the second socket, and measure $c(5)$.

Based on $T(1)$ and T_D, we determine the required speedup π'_D that achieves the deadline:

$$\pi'_D = \frac{T(1)}{T_D}$$

Because $T(1)$ is around 300 seconds, the required exploited parallelism is $\pi'_D = 3$, on a UMA core system with $n = 8$ cores.

From the trace of the operating system run-queue we determine the inherent parallelism $\pi(8, \infty)$ and the constrained parallelism $\pi(8, n)$ when n varies from one to eight. Using the measured values of $c(1)$, $c(2)$ and $c(5)$ as inputs, we apply our memory contention model to determine $\omega(n)$ for our program. With $\omega(n)$ and $\pi(8, \infty)$, we can apply the general parallelism performance model to predict the values of the inherent parallelism.

Because we are interested in finding out the configuration with the minimum number of active cores that meets the performance challenge, it is not needed to predict the exploited parallelism for all possible values of n. Because the $\pi'(n)$ might be non-monotonic with n, the range of n which is useful consists of values of n for which $\pi'(n)$ is growing. Using equation 3.19 the condition for which $\pi'(m, n)$ is increasing with n is (if $\omega(n) > -1$ and assuming there are enough threads, so $m \geq n$):

$$\frac{d\omega(n)}{dn} < \pi'(m, n) \cdot \frac{\partial \pi(m, n)}{\partial n} \tag{5.1}$$

Equation 5.1 expresses the range of useful values of n as the intervals for which

the rate of growth in memory overhead is smaller than the rate of growth in useful work. We use numerical differentiation to compute the values of n for which $\pi'(m,n)$ increases. The range of n that satisfies equation 5.1 is a union of j intervals $\cup_j[n_{min,j}, n_{max,j}]$. The number of cores of the best configuration is therefore $min\{n \in \cup_j[n_{min,j}, n_{max,j}] | \pi'(n) > \pi_D\}$

We apply numerical differentiation to compute the solutions of equation 5.1. The solution is $n \in [1,2] \cup [4,6]$. However, after applying the model for this set of configurations, the maximum speedup achieved is $\pi'(8,6) = 1.67$, which does not meet the parallelism performance deadline of $\pi'_d = 3$. For $n > 6$ the exploited parallelism decreases fast, because $\omega(n)$ grows rapidly with n.

For comparison purposes, we show in Figure 5.3 the measured values of the

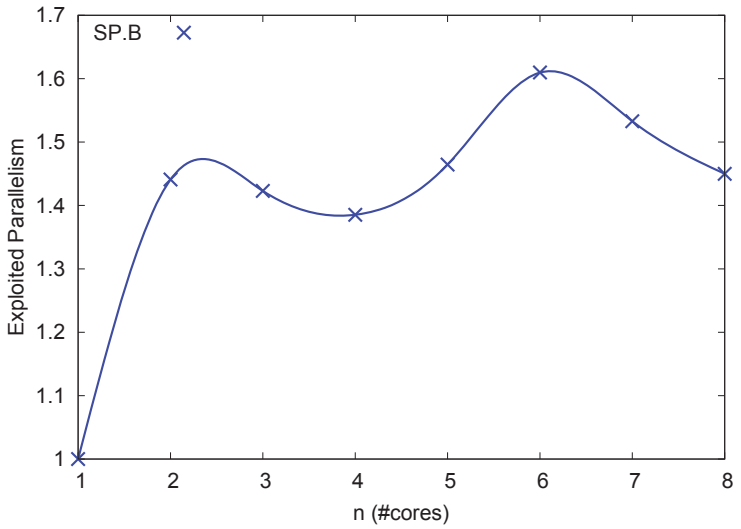


Figure 5.3: Measured exploited parallelism: *SP.B*

exploited parallelism when the number of active cores n ranges from one to eight, with spline smoothing of the predicted values. The ranges of n for which the speedup is increasing are correctly predicted as $[1,2] \cup [4,6]$ and the maximum measured speedup is $\pi'(8,6) = 1.60$, significantly below the required $\pi_D = 3$.

Therefore, the required execution time of under $T_D = 100$ seconds cannot be achieved on the target architecture, not even if all the cores in the system are used

at maximum frequency.

5.2.2 Understanding Parallelism Loss

Motivated by the poor parallelism performance of program *SP.B* on the target UMA multicore system, we apply the model to understand the sources of the parallelism loss. We study the parallelism loss due to data dependency and memory overhead for program *SP*.

Because a well known property of parallel programs is that speedup improves for larger problem sizes [47], we perform the analysis of the parallelism loss not only for problem size B, but also for three other sizes, W, A and C, to see if the changing the problem size affects the parallelism loss. The ratio of problem sizes W:A:B:C $\approx$ 1:4:16:64.

We predict the parallelism loss of *SP* when varying m from 1 to 8 threads. We have run the baseline run for predicting $\pi(m, \infty)$ and we derive $\omega(n)$ using measured $c(1)$, $c(4)$ and $c(5)$ on the target UMA system for all four problem sizes. Figure 5.4 shows the speedup loss for SP for all problem sizes. *SP* has a counter-intuitive behavior: *the speedup reduces as the problem size is increased*. We can see that small problem sizes lead to larger parallelism loss due to data dependency. This is expected considering that data dependency in *SP* is mostly induced by barriers [13]. Smaller input sizes lead to small intra-barrier times. Therefore, it is expected that problem W has data dependency as the most significant source of speedup loss. For larger inputs, data dependency reduces significantly. From the application of the model we observe that none of the problem sizes results leads to a value of exploited parallelism that meets the parallelism performance deadline $\pi'(m, n) > \pi_D$.

For large input sizes, the main source of speedup loss is the memory overhead, which increases both with the number of cores and with the problem size. For

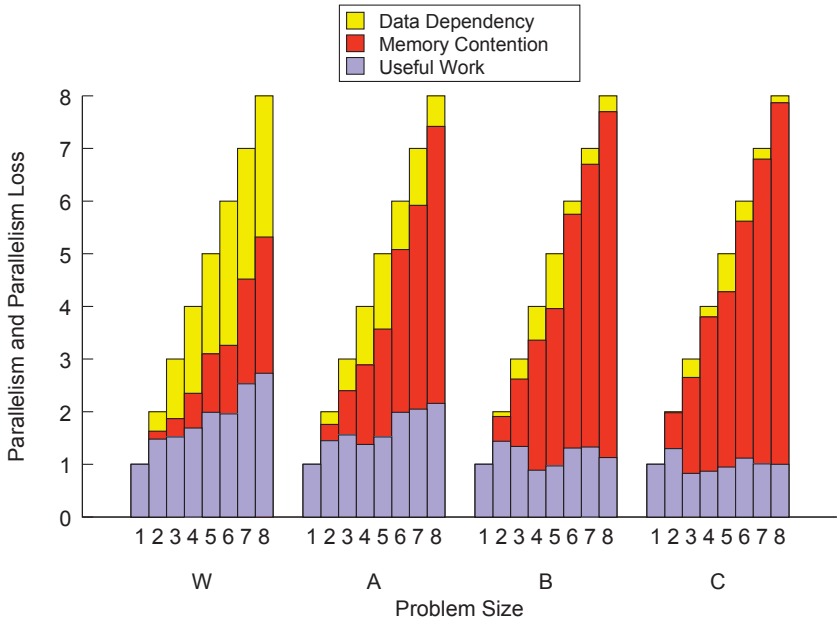


Figure 5.4: Modeled exploited parallelism and parallelism loss: *SP*

problem size C, $\pi'(m, n = m)$ degrades very close to 1, when $m > 2$, which means that allocating more than two cores to *SP.C* decreases speedup and increases number of resources used.

This analysis leads to two conclusions. Firstly, *SP* is particularly impacted by memory contention among cores, and therefore it should benefit from higher memory bandwidth, either through faster memory, larger memory bandwidth through NUMA architectures, or increased caching. Secondly, matching the input size to the number of cores should reduce significantly both execution time and number of cores required. The second implication is very important relative to energy use of the program. For program *SP*, on our target UMA system, the increase in n results in increase of both power usage and execution time. This results in a large increase in energy use.

5.2.3 Impact of Changing from UMA to NUMA

As the parallelism loss analysis suggests that memory contention among cores is the primary factor of performance loss, we apply our model to understand the effect of switching from a two-socket UMA system to a two-socket NUMA system. Specifically, we want to understand if the larger bandwidth in a NUMA system reduces the memory contention and improves the exploited parallelism to the point of meeting the parallelism performance deadline.

We derive the inherent parallelism for a two-socket quad-core NUMA, comparing the results with the two-socket quad-core UMA. On the NUMA system, we use a *fill-socket-first* core allocation policy, which means we want to fully utilize the cores on the first socket before we activate the cores on the second socket.

To determine $c(1)$, $c(2)$ and $c(5)$, we perform the set of baseline runs on a target dual-socket quad-core NUMA system. Figure 5.5 shows the modeled memory

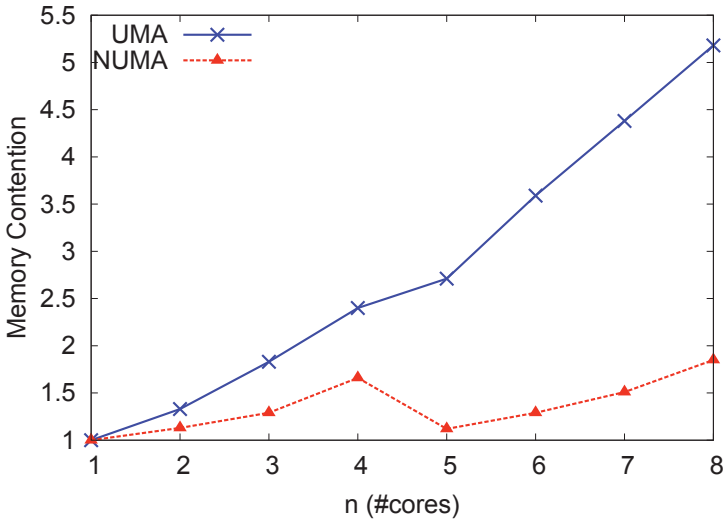


Figure 5.5: Modeled memory contention: *SP.B* on UMA & NUMA

contention $\omega(n)$ for different values of n in UMA and NUMA systems. For clarity, we show only program SP with problem size B. We observe that for values of n between one and four, memory contention grows on both UMA and NUMA

systems. However, there is a pronounced difference, even when only one memory node is used. Because our target NUMA system has significant higher memory bandwidth per node, due to more memory channels and larger cache size, the memory contention when using only one controller is at most $\omega(4) = 1.66$ on NUMA compared to $\omega(4) = 2.40$ on UMA. The memory contention drops when the second memory node is activated on NUMA. Overall, the memory contention is significantly lower on NUMA, $\omega(8) = 1.85$, compared to UMA $\omega(8) = 5.18$.

The effect of the lower memory contention on the exploited parallelism can be seen in Figure 5.6. The exploited parallelism on the NUMA system is larger compared to the UMA system. The maximum speedup increases more than two

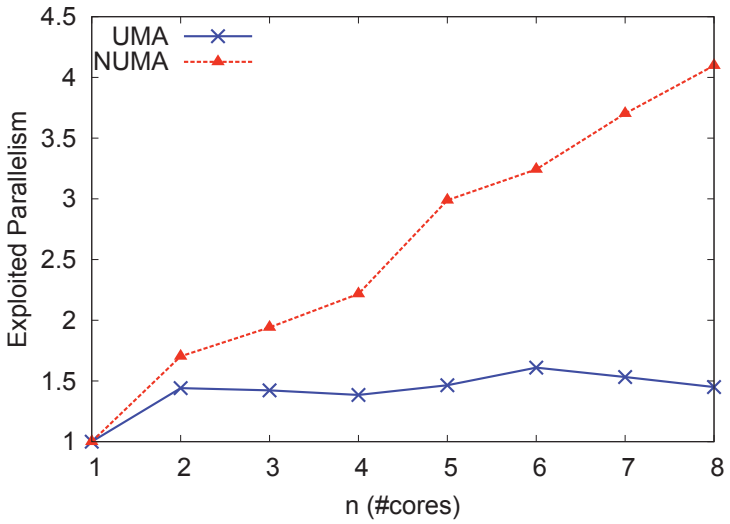


Figure 5.6: Modeled exploited parallelism: *SP.B* on UMA & NUMA

times, from the maximum value of $\pi'(8,6) = 1.60$ on UMA to $\pi'(8,8) = 4.09$ on NUMA. However, even on NUMA, program *SP.B* hits the memory wall, because more than 99% of the parallelism loss of the program is due to memory contention. To conclude the analysis, program *SP.B* meets the parallelism performance deadline of $\pi'_d = 3$ only on the NUMA system. The configuration with the minimum number of cores that achieves the target parallelism performance deadline is $n = 6$ with $\pi'(8,6) = 3.02$ on the target NUMA system.

5.3 Improving Energy Efficiency in Parallel Programs

5.3.1 Core-Frequency Configuration for Minimum Energy Use

An important aspect of parallel computing is the energy incurred by a program. While energy concerns have traditionally been associated with battery-powered embedded systems, the proliferation of datacenters that power today's interconnected computers have pushed these concerns to mainstream multicore servers.

In this section, we apply our model to predict the multicore configuration that achieves the minimum energy use for a program, and determine the relationship between performance and minimum energy usage. The system configuration is parameterized by the number of active cores, n, and the clock frequency of the cores, f.

We apply our modeling approach to predict the core-frequency configuration that achieves the minimum energy usage for three programs with different performance requirements: CPU-bounded using program EP, memory-bounded using program SP and I/O-bounded using program *memcached*. The inputs of the three programs are the same as used in the validation experiments shown in Chapter 4.

Figures 5.7, 5.8 and 5.9 show the execution time and total energy usage of EP, SP and *memcached* on the Exynos 4412 ARM Cortex-A9 system. Comparing the three program, the effects of the system bottleneck can be seen on the energy usage. For CPU-bounded programs such as EP, increasing the number of cores and core clock frequency does not increase the number of cycles incurred by the program. Thus, the execution time drops proportionally to the average number of active

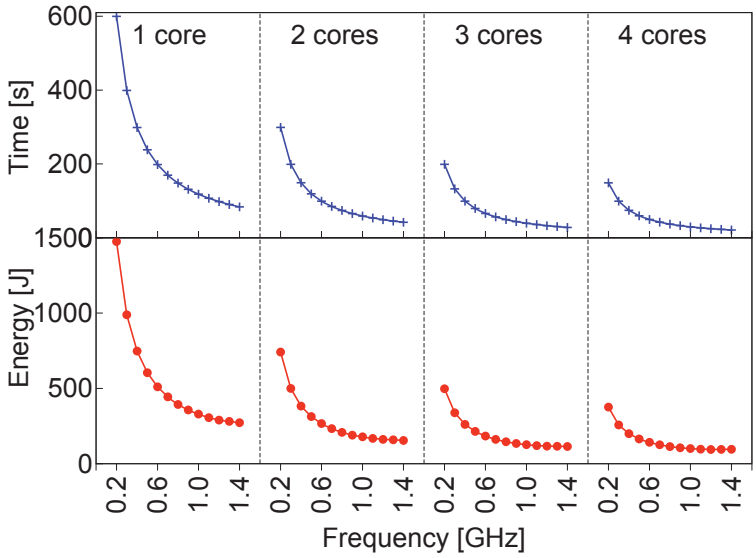


Figure 5.7: Execution time and energy usage of *EP*

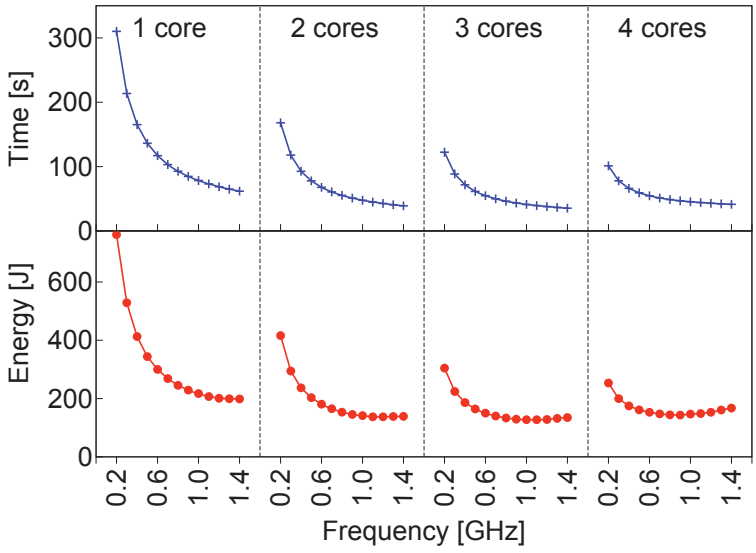


Figure 5.8: Execution time and energy usage of *SP*

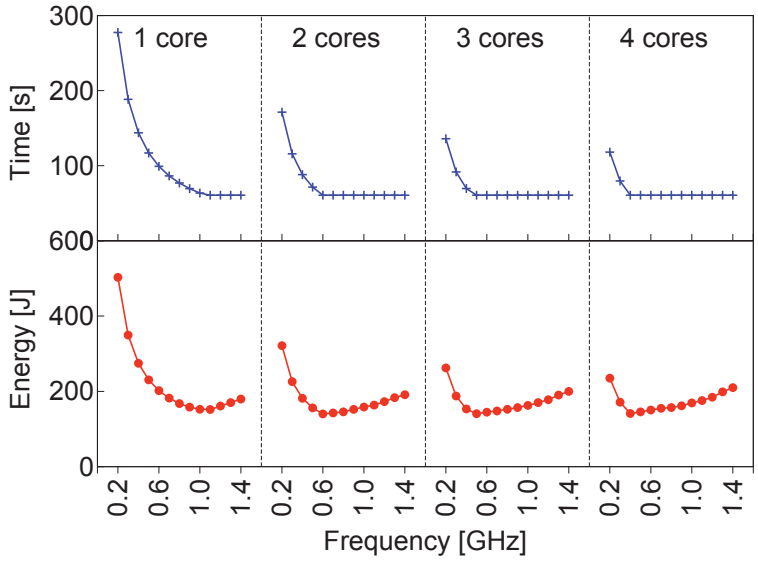


Figure 5.9: Execution time and energy usage of *memcached*

cores of the programs. Since EP is also embarrassingly-parallel, the execution time decreases linearly when the number of cores n and core clock frequency f increase. Even though the power usage increases superlinearly with the core clock frequency, the consistent reduction in execution time with an increase in n and f results in an overall decrease in energy usage as the number of cores and clock core frequency increase. Thus, for CPU-bounded programs, executing on high core counts and clock frequencies is advantages for both parallelism and energy, and the minimum execution time is reached on a configuration of n=4 cores and f=1.4 GHz.

Memory-bounded programs such as SP are less energy-efficient than CPU-bounded applications. Because the response time of the program is dictated by the memory response time, the total number of cycles of the program will increase when the core clock frequency increase (as modeled by equation 3.28, and as can be seen in the validation section in Figure 4.13(a). Furthermore, the slope of the cycles over core frequency f is the memory response time. As the number of cores increase, the slope becomes steeper due to the memory contention among cores. Because of the increase in number of cycles with n and f, executing on

large number of cores limits the decrease in execution time. But power usage still increases with n and f. Thus, the energy usage of the program increases for large cores counts and clock frequencies, but without a proportional reduction in execution time. The minimum execution time is reached on a configuration with n=3 cores and f=1.4 GHz, while the minimum energy usage is reached when executing on n=3 cores and f=1.1 GHz. Because of the memory boundedness, executing on four cores is contraindicated, as it increases both execution time and energy usage.

Next we comment on the I/O-bounded program *memcached*. Because the execution time is lower-bounded by the I/O response time, increasing the core clock frequency and core counts does not translate into a reduction of execution time, as shown in the validation Figure 4.13(d). On the Exynos 4412 ARM Cortex-A9 system, memcached saturates the I/O bandwidth on n=2 cores at f=600 MHz. Increasing the core counts and frequency past this point is completely inefficient, as the power use and incurred energy increase without any reduction in execution time.

Table 5.3 shows the configuration that achieves the minimum execution time, and the minimum energy usage for all programs studied. Only CPU-bounded

Program	**Bottleneck**	**Configuration**			
		Min. Time		**Min. Energy**	
		n	f **[GHz]**	n	f **[GHz]**
EP	Cores	4	1.4	4	1.4
BT	Memory	4	1.4	4	1.1
CG	Memory	3	1.4	3	1.0
FT	Memory	3	1.4	3	1.2
IS	Memory	3	1.4	3	1.0
SP	Memory	3	1.4	3	1.1
blackscholes	Cores	4	1.4	4	1.4
x264	Cores	4	1.4	4	1.4
memcached	I/O	2	0.6	2	0.6

Table 5.3: Minimum time and energy configurations

Program	Time [s]					Energy [J]				
	Model	Ondemand	Saving [%]	Powersave	Saving [%]	Model	Ondemand	Saving [%]	Powersave	Saving [%]
BT	43.51	43.51	0.0	165.97	281.5	169.47	183.11	8.0	418.46	146.9
CG	40.61	46.74	15.1	104.0	156.1	140.4	187.08	33.2	259.65	84.9
FT	11.57	14.77	27.7	42.46	266.9	43.25	60.24	39.3	106.46	146.1
IS	52.13	55.91	7.3	140.09	168.7	185.9	226.72	22.0	351.34	89.0
SP	35.34	41.31	16.9	101.04	185.9	127.35	167.28	31.4	253.34	98.9
memcached	60.63	60.63	0.0	118.25	95.0	140.07	209.95	49.9	235.11	67.9

Table 5.4: Execution time and energy savings over Linux DVFS policies

programs execute efficiently on large number of cores and high core frequency. For memory and I/O bounded programs, the minimum energy usage is achieved when using a frequency lower than the maximum, and only two or three of the four cores.

Next we quantify the execution time improvements and energy savings that can be achieved when using a core-frequency dictated by our model, versus the default core-frequency policies employed by Linux. By default, many Linux systems use the *ondemand* dynamic voltage and frequency scaling policy, which applies the maximum frequency if the cores are utilized and minimum frequency when the cores are not utilized. Our analysis shows that this policy would lead to energy wastage for execution of memory-bounded programs on any number of cores. Moreover, the *powersave* policy of keeping the core frequencies at the minimum setting increases the execution time dramatically, to the point that energy usage is disproportionately high. Table 5.4 shows the execution time improvements and energy savings enabled by core-frequency configurations predicted by our model for memory and I/O bounded programs, versus the *ondemand* and *powersave* policies of Linux DVFS governors. The savings are expressed as percentage, and are computed as the difference between the optimal configuration predicted by the model, relative to the configuration using n=4 cores under *ondemand* and *powersave* DVFS governors.

This analysis shows that for memory or I/O bounded programs, an efficient execution operates neither at the highest frequency nor using the full core counts.

Compared to the Linux *ondemand* DVFS policy, our model predicts configurations that balance the cores, memory and I/O resources and results in a reduction of execution time and energy usage by up to 27% and 50%, respectively. Compared to the *powersave* policy, our model reduces execution time by a factor of 2.8 and energy usage by up to 1.5 times.

5.3.2 When is Low Power not Energy-Efficient?

There has always been a trade-off between reducing power usage and and increasing the performance of processing systems. Many currently-available ARM Cortex-A9 systems have been configured mostly for mobile computing devices such as phones or tablets, and thus their resources are sized for the balance between cores, memory and I/O required by mobile applications. As part of this balance, they are provisioned with lower *achievable memory bandwidth* than traditional x64 server systems. For example, the memory-level parallelism in Cortex-A9 chips is limited to two outstanding memory requests [84], as compared to ten in Intel chips [86]. Because mobile apps are typically not memory-bounded, the size of the caches range between 256 kB and 1 MB in the commodity Cortex-A9 systems shipped by vendors such as Samsung, Nvidia, Texas Instruments or ST-Ericsson. In contrast, the size of the cache memory in most x64 server systems exceeds 10 MB. Furthermore, the memory subsystem in many currently available ARM Cortex-A9 chips use low power memory, with 32-bit data bus width and operating at lower clock frequencies compared with traditional memory chips. As a result of these factors, low-power computing on ARM Cortex-A9 might suffer from a larger imbalance between arithmetic and memory performance than traditional x64 systems. Considering that many types of server workloads are I/O or memory-intensive, the large gap between core and memory performance might lead to unexpected results.

To address this difference between core and memory performance in traditional and low-power multicore systems, we compare the execution of memory-bounded *SP* on low-power Cortex-A9 with the Intel NUMA multicore system.

We ran program *SP* with 400 iterations on a grid of 162^3 (similar to input size *C*, but with more iterations). The input results in a working set large enough to exceed the caches of both systems, but fits into 1 GB of main memory.

We apply our prediction for execution time on the low-power multicore for all core-frequency configurations, and observe that the minimum execution time exceeds 17,000 seconds. In contrast, the execution time on the x64 system is 330 seconds, using all cores at maximum frequency. The large gap between execution times appears because the Intel system is equipped with much larger caches, and thus, incurs 15 times less cache misses. Furthermore, the main memory bandwidth is around 8 times higher, while the bandwidth of the caches are ten times (for L1) and four times (when comparing Intel L3 to ARM L2) higher. The average power used by the Intel system (with disks turned off) is around 210W.

We extend this analysis for datacenters, factoring the additional power required for power conversion and cooling the systems. The Power Usage Effectiveness (PUE) of a datacenter measures the total power required to deliver 1 Watt of IT power. From literature, we identify two PUE bounds: the lower bound is PUE=1.13, in a Google datacenter [5], while an the upper bound is 1.89 [6]. Figure 5.10 shows the predicted values of energy consumption on low-power multicore, for all configurations of cores-frequency, compared with an execution on all x64 cores at maximum frequency, for PUE between 1.13 and 1.89. Few ARM configurations manage to achieve lower energy cost than x64, with less than 7% energy reduction, but at a cost of incurring execution time more than 50× higher. However, the x64 execution is not energy efficient because all cores are used at maximum frequency even though the memory is the bottleneck. We conclude that

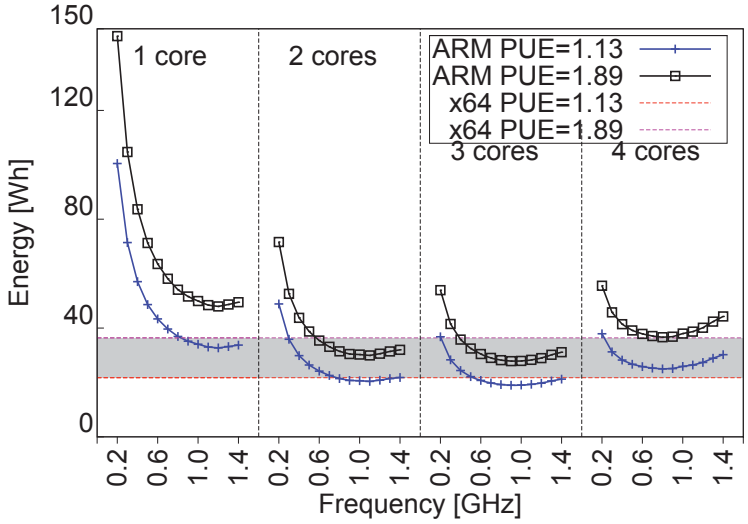


Figure 5.10: *SP* energy usage: ARM Cortex-A9 vs Intel NUMA x64

memory-bounded programs can be unsuitable for low-power multicores, even if optimizing the core-frequency configuration to achieve best performance at minimum energy cost.

This analysis shows that the balance between core and memory performance plays an important role in achieving energy-efficient executions. Thus, the design of the Exynos 4412 ARM Cortex-A9, which favors high CPU processing power but comparatively little memory bandwidth leads to energy wastage. The question then becomes: how should the system be balanced? By turning off under-utilized cores or by increasing the performance of the memory and I/O subsystem?

5.3.3 Improving Energy Efficiency of Low-power Multicore Systems

ARM multicores typically have good energy efficiency when used as mobile computers, due to their sleep states and low-power operation [84]. However, our previous analysis shows that resource imbalances lead to large energy wastage in server workloads. Thus, leveraging on the idea that ARM systems are highly con-

figurable, we apply our model to understand how to lower the energy usage of server workloads.

As the key to improving energy proportionality is system balance, we apply our model to predict the performance of program *memcached* under different hardware configurations that balance the system resources. Figure 5.11 shows the response times of different resources for the original hardware configuration (100 Mbps Ethernet, one memory controller), when using two active cores. This number of cores is selected as it achieves the best performance at minimum energy cost. For

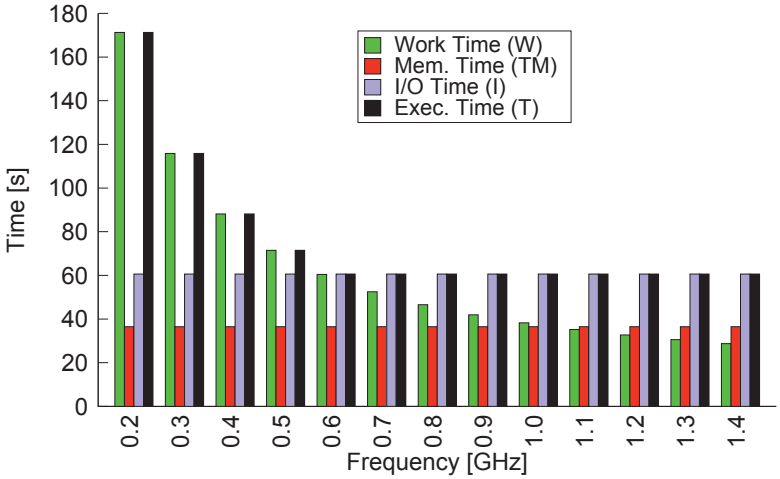


Figure 5.11: *Memcached* Response Times

small core frequencies, the CPU work time ($W = \frac{w+b}{f}$) is the bottleneck, but at 600 MHz the CPU response time matches the I/O response time. Beyond 600 MHz, the I/O bandwidth becomes the bottleneck, and the execution time does not reduce anymore.

This analysis allows significant energy savings if two cores are turned off. Unfortunately, the operating system on our system does not allow selective power off of a subset of cores, and thus we cannot measure directly the energy savings. However, using figures from related work [67, 84], by shutting down two out of the four cores we can estimate a reduction of processor power between a conservative

25% and an optimistic 50%. With these figures, applying the configuration predicted by our model allows for a reduction in total energy savings between 13% and 31%, and without compromising the execution time performance.

We analyze the performance impact of replacing two system components. First, the 100 Mbps Ethernet is replaced with 1 GBps Ethernet, without modifying any other component. In this analysis, we consider a Gigabit Ethernet adapter with a power consumption of 600 mW, which is typical for a power-efficient network card. The I/O time for *memcached* is composed of transfer time $I_T = 56$ s and blocking time $T_B = 5.1$ s. With a 1 Gbit Ethernet, the total I/O time becomes $I = 10.7$ s. However, because the I/O device is memory mapped, we consider that the Gigabit Ethernet will utilize 125 MB/s out of the approximately 800 MB/s memory bandwidth. Thus, s_M increases from 38 ns to 45 ns. Applying Equation 3.41, T_M increases from 36.5 s to 47 s. Thus, the effect of moving to 1 GBit Ethernet is a reduction from of execution time from $T = 61$ s to $T = 47$ s, and the system bottleneck becomes the memory. Due to the increase in I/O power by 400 mW and due to the increase in stall cycles, the average power increases by approximately 500 mW. Figure 5.12 shows that total energy decreases by switching

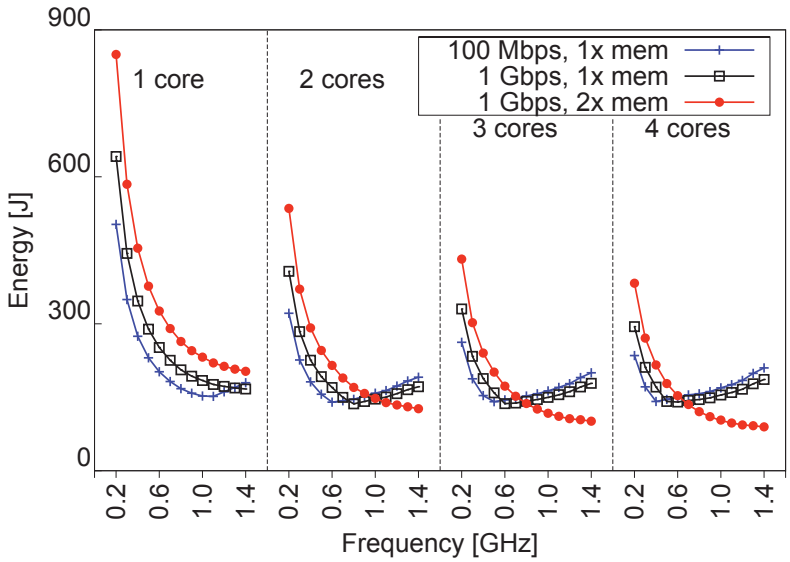


Figure 5.12: *Memcached* with 1 Gbit Ethernet and Double Memory Bandwidth

to a 1 Gbps Ethernet because the decrease in execution time offsets the increase in average power. Since the new bottleneck is the memory, we consider next the impact of doubling the effective memory bandwidth (ARM Cortex-A9 systems can be configured to up to quad-memory channels, while the next generations ARM Cortex-A15 and ARM Cortex-A50 support more outstanding memory requests and can be configured to use LPDDR3). With the double memory bandwidth, the memory response time drops to 18.1 s, and the system bottleneck becomes the core. We consider a pessimistic scenario, where power consumption is the quadruple of the original memory system (100 mW idle memory power and 1 W active memory power). However, the energy consumption still decreases by more than 50%, and *memcached* becomes CPU-bounded. It achieves best performance and minimum energy when using all cores at full frequency.

This analysis showed that reducing the imbalance among core, memory and I/O leads to lower energy usage. However, counter to intuition, we showed that balancing the resources by adding *more hardware resources* results in higher energy savings (50% reduction compared to the original hardware configuration), compared to balancing the resources by turning off underutilized cores (13-31% reduction). However, this results in higher average power consumption.

Our model suggest that future energy-efficient multicore system must focus on improving execution time by improving the memory and I/O subsystems, rather than to limit performance because of adhering to a low-power design. While this has important implications to circuit design, thermal and power management, our analysis concludes that future energy-efficient systems must take on these challenges, and move away from a reliance on low-power to achieve energy-efficiency.

Considering that next-generation ARM systems such as ARM Cortex-A15 and the 64-bit ARM Cortex-A50 family target much improved memory and I/O subsystems [112], we conclude that our analysis validates the direction that the indus-

try is following. If the trends continue, future multicore systems based on ARM multicores will deliver better energy-efficiency than current ARM Cortex-A9, but at a cost of higher power usage.

5.4 Summary

In this chapter we applied the model to understand the parallelism-energy performance of workloads covering HPC, financial, multimedia and datacenter workloads. We showed that many applications lose significant parallelism, and this contributes to an increase in both execution time and energy usage. Furthermore, we showed that analyzing the parallelism loss of a program is a better handle at optimizing the energy usage of a program, compared to analyzing just the energy proportionality of the program execution.

Following this analysis we described three scenarios in which the proposed parallelism-energy performance analysis framework is used to drive performance optimization for both execution time and energy usage. First we showed that our model can be used to determine the minimum number of cores that achieves a required execution time on a commodity multicore server. If the system cannot achieve the desired execution time, we showed that our model can be applied to understand the exploited parallelism and parallelism loss due to data dependency and memory contention, and to analyze the impact of switching from UMA to NUMA memory interconnect. Second, we predict the optimal core frequency and number of active cores that achieves the minimum energy utilization in low power ARM multicores and show that 50% energy savings can be made using predictions of our model, compared with Linux *ondemand* and up to 1.5× compared to the *powersave* frequency governors. Finally, we show that the energy consumption of low-power multicores when executing server workloads can be reduced if the

performance of the memory and the I/O subsystems is increased, even if this leads to higher power usage.

Chapter 6

Conclusions

We conclude the thesis by presenting a summary of our main contributions followed by a discussion on the further research directions.

6.1 Thesis Summary

This thesis presents an approach for understanding the parallelism and energy performance of shared-memory programs on both traditional and low-power multicore systems. Our analytical models for parallelism and energy performance, are driven by insights derived from measurements, with inputs from traces of operating system run-queue, hardware events counters and static power measurements.

Our parallelism model predicts the inherent parallelism and exploited parallelism of a program, and quantifies the parallelism loss due to data-dependency and memory contention. Validation of the model was performed against measurements using HPC workloads from NPB suite, financial and multimedia workloads from PARSEC benchmarks and datacenter workloads such as *memcached* on state-of-the-art UMA and NUMA systems with up to 48 cores. Our model results differ from measurements by around 6%-13% for a range of traditional Intel/AMD and

low-power multicore systems. Our energy performance model is designed to understand the relationship between parallelism and energy performance. Given a shared-memory program, the energy model predicts the average power required, the total energy usage and the *knee clock frequency* where core and memory performance are balanced, and execution time and energy performance are close to optimum.

In carrying out extensive measurement analysis on which the model is based, we draw a number of insights on memory contention in multicore systems [117]. In contrast with previously reported observations [63], we show that the memory traffic is not always bursty, and memory burstiness depends on problem size. Small problem size do not generate memory contention, but exhibit bursty memory traffic. In contrast, large parallel programs which generate significant memory contention are observed to cause less bursty traffic. This simplifies the modeling of the memory contention in large multicore systems. For problems with large contention, we show that a single-server queueing system accurately models the behavior of the memory system [115].

Our modeling approach is used to drive performance optimizations for different user scenarios. For users that are concerned with program executions under a strict deadline, we show that our models can be used to determine the multicore configuration that achieves a required execution time. Counter to intuition, if the program exhibits severe memory contention, the smallest execution time is achieved using low core counts [114, 116]. If the desired execution time cannot be met, the parallelism loss from data dependency and memory contention, as predicted by our model, can assist the user in making appropriate changes to the program or the machine. For example, we show that switching from a UMA system to a NUMA system is the most effective configuration change that reduces contention and improves speedup.

For users of parallel programs that execute under tight energy constraints, we show that our model predicts the number of cores and clock frequency that optimizes the energy use without increasing the response time of the program. Furthermore, we show that in memory-bounded programs executing on low-power ARM multicore systems, our model predicts configurations that balance the cores, memory and I/O resources and results in a reduction of execution time and energy usage by up to 27% and 50%, respectively. Compared to the *powersave* policy, our model reduces execution time by a factor of 2.8 and energy usage by up to 1.5 times.

Finally, by modeling the relationship between performance and energy, this thesis suggests that the key for improving energy efficiency of multicore systems is to balance the cores, memory and I/O resources, such that waiting time and idle energy in the system are minimized. For a program execution that leads to a imbalance among resources, we show that restoring system balance by increasing the performance of the bottleneck devices leads to lower execution time and energy usage, compared to turning off under-utilized resources, even if this is achieved with higher power usage [116].

6.2 Future Research Directions

Our proposed approach of modeling the time and energy performance via exploited parallelism and the parallelism loss due the data-dependency, memory contention and I/O overheads can be extended in several directions.

A first extension involves extending the model to distributed-memory programs. For such programs, the model needs to be extended in two ways. First, a model for the communication overhead needs to be included, such that we can quantify the parallelism loss when processes are waiting among themselves. Sec-

ond, the overlap between the three response times among different nodes needs to be modeled. For example, in a datacenter workload that executes on multiple multicore server nodes, the response times of a job will depend on the slowest node, and the response time on the slowest node in turn depends on the overlap between its cores, memory and I/O service times.

Another direction is to tackle the increasing presence of heterogeneous multicore systems. First, heterogeneity among the cores of a server is becoming the norm in the dark silicon era. However, the question of whether intra-node heterogeneity leads to a more efficient usage of the off-chip resources is still unanswered. For example, the next generation ARM big.LITTLE architecture, such as Samsung Exynos 5 couples four low-power in-order ARM Cortex-7 cores with four high-performance out-of-order ARM Cortex-A15 cores. Because of their in-order execution, executions on ARM Cortex-A7 might not fully exercise the off-chip memory bandwidth or the I/O resources. Because these off-chip resources still consume significant power even when under-utilized, it is thus possible that significant energy is still wasted. On the other hand, executions on the ARM Cortex-A15 cores might still be imbalanced, if the off-chip resources are bottleneck before the cores reach their full execution capacity. An optimal solution might involve a controlled transition between the two types of cores, supported by a power allocation model that matches the on-chip and off-chip resource supply to the workload demands.

Secondly, heterogeneity among nodes of a cluster can be exploited for energy-efficient executions of datacenter workloads. Such workloads must often obey strict response time constraints. Thus, to maintain a good user experience, the service time inside a datacenter might be different among users, even for the same type of job. While a system with only low-power nodes may not meet a target deadline, a system using only high-performance nodes may require an inordinate

amount of energy when operating at higher performance level than necessary. Ideally, a system should allow a range of configurations that decreases the energy progressively as the deadline is relaxed. This motivates the case for analyzing a heterogeneous cluster system with a mix of high-performance nodes and low-power nodes.

Lastly, to support performance analysis of data-intensive programs, the extension to distributed computing can be supplemented by a cost model for data movement among the nodes of a cluster and the cost of storage I/O.

References

[1] *Advanced Configuration and Power Interface*. http://www.acpi.info/.

[2] *Go Programming Language*. http://golang.org/.

[3] *JEDEC STANDARD Low Power Double Data Rate 2 (LPDDR2)*. http://www.jedec.org/sites/default/files/docs/JESD209-2B.pdf.

[4] *Memcached*. http://memcached.org/.

[5] *Google Data Center Efficiency: How We Do It*, Oct 2012. http://www.webcitation.org/6C8PjIMYd.

[6] *Uptime Institute 2012 Survey*, Oct 2012. http://uptimeinstitute.com/2012-survey-results/.

[7] D. Abts, M. R. Marty, P. M. Wells, P. Klausler, and H. Liu. Energy Proportional Datacenter Networks. *Proc. of 37th Annual International Symposium on Computer Architecture*, pages 338–347, 2010.

[8] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. *Proc. of AFIPS Spring Joint Computer Conference*, pages 483–485, Atlantic City, USA, 1967.

[9] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. Fawn: a fast array of wimpy nodes. *Proc. of ACM SIGOPS*

22nd Symposium on Operating Systems Principles, SOSP '09, pages 1–14, Big Sky, Montana, USA, 2009.

[10] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006.

[11] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. *Proc. of ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, pages 53–64, London, UK, 2012.

[12] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35:59–67, 2002.

[13] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks — Summary and Preliminary Results. *Proc. of 3rd ACM/IEEE Conference on Supercomputing*, pages 158–165, Albuquerque, USA, 1991.

[14] M. K. Bane and G. D. Riley. Extended Overhead Analysis for OpenMP (Research Note). *Proc. of the 8th International Euro-Par Conference on Parallel Processing*, pages 162–166, London, UK, 2002.

[15] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, and M. Schulz. A Regression-based Approach to Scalability Prediction. *Proc. of 22nd International Conference on Supercomputing*, pages 368–377, Island of Kos, Greece, 2008.

[16] L. A. Barroso and U. Hölzle. The Case for Energy-Proportional Computing. *Computer*, 40(12):33–37, Dec. 2007.

[17] R. Bertran, M. Gonzalez, X. Martorell, N. Navarro, and E. Ayguade. Decomposable and Responsive Power Models for Multicore Processors Using Performance Counters. *Proc. of 24th ACM International Conference on Supercomputing*, pages 147–158, Tsukuba, Japan, 2010.

[18] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. *Proc. of 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, Toronto, Canada, Oct. 2008.

[19] R. D. Blumofe and C. E. Leiserson. Space-efficient Scheduling of Multithreaded Computations. *Proc. of 25th Annual ACM Symposium on Theory of Computing*, pages 362–371, San Diego, USA, 1993.

[20] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011.

[21] J. M. Bull. A Hierarchical Classification of Overheads in Parallel Programs. *Proc. of 1st IFIP TC10 International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 208–219, Berlin, Germany, 1996.

[22] A. Carroll and G. Heiser. An Analysis of Power Consumption in a Smartphone. *Proc. of 11th USENIX Annual Technical Conference*, Boston, USA, 2010.

[23] A. Carroll and G. Heiser. The Systems Hacker's Guide to the Galaxy Energy Usage in a Modern Smartphone. *Proc. of 4th Asia-Pacific Workshop on Systems*, pages 5:1–5:7, Singapore, Singapore, 2013.

[24] J. Chen, M. Dubois, and P. Stenström. SimWattch: Integrating Complete-System and User-Level Performance and Power Simulators. *IEEE Micro*, 27:34–48, 2007.

[25] S. Cho and L. Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. *Proc. of 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 455–468, Orlando, USA, 2006.

[26] K. Choi, R. Soma, and M. Pedram. Dynamic Voltage and Frequency Scaling Based on Workload Decomposition. *Proc. of 10th International Symposium on Low Power Electronics and Design*, pages 174–179, Newport Beach, USA, 2004.

[27] M. Curtis-Maury, F. Blagojevic, C. D. Antonopoulos, and D. S. Nikolopoulos. Prediction-Based Power-Performance Adaptation of Multithreaded Scientific Codes. *IEEE Transactions on Parallel and Distributed Systems*, 19(10):1396–1410, 2008.

[28] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz. Prediction Models for Multi-dimensional Power-performance Optimization on Many Cores. *Proc. of 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 250–259, Toronto, Canada, 2008.

[29] J. D. Davis, J. Laudon, and K. Olukotun. Maximizing CMP Throughput with Mediocre Cores. *Proc. of 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 51–62, 2005.

[30] J. Donald and M. Martonosi. Techniques for Multicore Thermal Management: Classification and New Exploration. *SIGARCH Computer Architecture News*, 34:78–88, 2006.

[31] A. B. Downey. A Model For Speedup of Parallel Programs. Technical Report UCB/CSD-97-933, EECS Department, University of California, Berkeley, 1997.

[32] A. B. Downey. A Parallel Workload Model and its Implications for Processor Allocation. *Proc. of 6th IEEE International Symposium on High Performance Distributed Computing*, 1997.

[33] J. E. G. Coffman, M. R. Garey, and D. S. Johnson. An Application of Bin-Packing to Multiprocessor Scheduling. *SIAM Journal on Computing*, 7(1):1–17, 1978.

[34] D. Eager, J. Zahorjan, and E. Lazowska. Speedup Versus Efficiency in Parallel Systems. *IEEE Transactions on Computers*, 38(3):408–423, 1989.

[35] N. El-Sayed, I. A. Stefanovici, G. Amvrosiadis, A. A. Hwang, and B. Schroeder. Temperature Management in Data Centers: Why Some (Might) Like it Hot. *Proc. of ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, London, UK, 2012.

[36] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. *Proc. of 38th Annual International Symposium on Computer Architecture*, pages 365–376, 2011.

[37] A. Fedorova, M. Seltzer, and M. D. Smith. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. *Proc. of 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 25–38, Brasov, Romania, 2007.

[38] S. Fuller and L. Millett. Computing Performance: Game Over or Next Level? *Computer*, 44(1):31–38, Jan.

[39] K. Fürlinger and M. Gerndt. ompP: A Profiling Tool for OpenMP. *Proc. of 1st International Workshop on OpenMP*, Eugene, USA, 2005.

[40] K. Fürlinger and M. Gerndt. Analyzing Overheads and Scalability Characteristics of OpenMP Applications. *Proc. of the 7th International Meeting on High Performance Computing for Computational Science*, Rio de Janeiro, Brazil, 2006.

[41] A. Ganapathi, K. Datta, A. Fox, and D. Patterson. A Case for Machine Learning to Optimize Multicore Performance. *Proc. of 1st USENIX Conference on Hot Topics in Parallelism*, Berkeley, USA, 2009.

[42] A. Gandhi, M. Harchol-Balter, and I. J. B. F. Adan. Server Farms with Setup Costs. *Performance Evaluation Review*, 67(11):1123–1138, 2010.

[43] A. Gandhi, M. Harchol-Balter, R. Das, and C. Lefurgy. Optimal Power Allocation in Server Farms. *Proc. of ACM SIGMETRICS/Performance Joint Conference on Measurement and Modeling of Computer Systems*, pages 157–168, 2009.

[44] A. Gandhi, M. Harchol-Balter, and M. A. Kozuch. Are Sleep States Effective in Data Centers? *Proc. of 3rd International Green Computing Conference*, pages 1–10, 2012.

[45] R. Ge, X. Feng, and K. W. Cameron. Modeling and Evaluating Energy-performance Efficiency of Parallel Processing on Multicore Based Power Aware Systems. *Proc. of 23rd IEEE International Symposium on Parallel&Distributed Symposium*, 2009.

[46] M. Geimer, F. Wolf, B. J. N. Wylie, E. Ábrahám, D. Becker, and B. Mohr. The Scalasca Performance Toolset Architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.

[47] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31(5):532–533, 1988.

[48] E. Haritan, T. Hattori, H. Yagi, P. Paulin, W. Wolf, A. Nohl, D. Wingard, and M. Muller. Multicore Design is the Challenge! What is the Solution? *Proc. of 45th Annual Design Automation Conference*, pages 128–130, Anaheim, California, 2008.

[49] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2006.

[50] A. Herdrich, R. Illikkal, R. Iyer, D. Newell, V. Chadha, and J. Moses. Rate-based QoS Techniques for Cache/Memory in CMP Platforms. *Proc. of 23rd International Conference on Supercomputing*, pages 479–488, Yorktown Heights, USA, 2009.

[51] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, 2008.

[52] U. Hölzle. Brawny cores still beat wimpy cores, most of the time. *IEEE Micro*, 30(4), 2010.

[53] R. Hood, H. Jin, P. Mehrotra, J. Chang, J. Djomehri, S. Gavali, D. Jespersen, K. Taylor, and R. Biswas. Performance Impact of Resource Contention in Multicore Systems. *Proc. of 24th IEEE International Parallel & Distributed Processing Symposium*, Atlanta, USA, 2010.

[54] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability, Programmability*. McGraw-Hill Higher Education, 1992.

[55] C. Iancu, S. Hofmeyr, F. Blagojevic, and Y. Zheng. Oversubscription on Multicore Processors. *Proc. of 24th IEEE International Symposium on Parallel & Distributed Processing*, Atlanta, USA, 2010.

[56] E. Ipek, B. R. de Supinski, M. Schultz, and S. A. McKee. An Approach to Performance Prediction for Parallel Applications. *Proc of 11th International Euro-Par Conference on Parallel Processing*, pages 196–205, Monte de Caparica, Portugal, 2005.

[57] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. *Proc. of 35th Annual International Symposium on Computer Architecture*, pages 39–50, Beijing, China, 2008.

[58] C. Isci and M. Martonosi. Runtime Power Monitoring in High-End Processors: Methodology and Empirical Data. *Proc. of 36th Annual IEEE/ACM International Symposium on Microarchitecture*, San Diego, USA, 2003.

[59] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. Wiley, 1991.

[60] N. K. Jha. Low Power System Scheduling and Synthesis. *Proc. of IEEE/ACM International Conference on Computer-Aided Design*, pages 259–263, San Jose, California, 2001.

[61] A. Karbowski. Amdahl's and Gustafson-Barsis Laws Revisited. *CoRR*, abs/0809.1177, 2008.

[62] T. Karkhanis and J. E. Smith. A Day in the Life of a Data Cache Miss. *Proc. of 2nd Workshop on Memory Performance Issues*, Anchorage, USA, 2002.

[63] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers. *Proc. of 16th International Symposium on High Performance Computer Architecture*, Bangalore, India, 2010.

[64] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Caşcaval. How Much Parallelism Is There in Irregular Applications? *Proc. of 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Raleigh, USA, 2009.

[65] M. Kumar. Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications. *IEEE Transactions on Computers*, 37(9):1088–1098, 1988.

[66] W. Lang, S. Harizopoulos, J. M. Patel, M. A. Shah, and D. Tsirogiannis. Towards Energy-Efficient Database Cluster Design. *PVLDB*, 5(11):1684–1695, 2012.

[67] E. Le Sueur and G. Heiser. Slow Down or Sleep, that is the Question. *Proc. of USENIX Annual Technical Conference*, Portland, USA, 2011.

[68] J. Lee, V. Sathisha, M. Schulte, K. Compton, and N. S. Kim. Improving Throughput of Power-Constrained GPUs Using Dynamic Voltage/Frequency and Core Scaling. *Proc. of the 20th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–120, Minneapolis, USA, 2011.

[69] K.-J. Lee and K. Skadron. Using Performance Counters for Runtime Temperature Sensing in High-Performance Processors. *Proc. of 19th IEEE International Parallel & Distributed Processing Symposium Workshops*, Denver, USA, 2005.

[70] T. Li and L. K. John. Run-time Modeling and Estimation of Operating System Power Consumption. *SIGMETRICS Performance Evaluation Review*, 31:160–171, 2003.

[71] W.-Y. Liang, S.-C. Chen, Y.-L. Chang, and J.-P. Fang. Memory-aware Dynamic Voltage and Frequency Prediction for Portable Devices. *Proc. of 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Kaohsiung, Taiwan, 2008.

[72] M. Lin, A. Wierman, L. Andrew, and E. Thereska. Dynamic Right-sizing for Power-proportional Data Centers. *Proc. of 30th IEEE International Conference on Computer Communications*, pages 1098–1106, 2011.

[73] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the Last Line of Defense Before Hitting the Memory Wall for CMPs. *Proc. of 10th International Symposium on High Performance Computer Architecture*, pages 373–380, San Jose, USA, 2004.

[74] F. Liu, X. Jiang, and Y. Solihin. Understanding How Off-chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance. *Proc. of 16th International Symposium on High Performance Computer Architecture*, Bangalore, India, 2010.

[75] G. Long, D. Fan, and J. Zhang. Characterizing and Understanding the Bandwidth Behavior of Workloads on Multi-core Processors. *Proc. of 15th*

International European Conference on Parallel and Distributed Computing, pages 110–121, Delft, The Netherlands, 2009.

[76] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi. Scale-out Processors. *Proc. of 39th Annual International Symposium on Computer Architecture*, 2012.

[77] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *Proc. 23th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, Chicago, USA, 2005.

[78] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *Computer*, 35:50–58, 2002.

[79] Z. Majo and T. R. Gross. Matching Memory Access Patterns and Data Placement for NUMA Systems. *Proc. of 10th International Symposium on Code Generation and Optimization*, pages 230–241, 2012.

[80] K. Malladi, F. Nothaft, K. Periyathambi, B. Lee, C. Kozyrakis, and M. Horowitz. Towards energy-proportional datacenter memory with mobile dram. *Proc. of 39th Annual International Symposium on Computer Architecture*, pages 37–48, Portland, USA, 2012.

[81] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz. Towards Energy-proportional Datacenter Memory with Mobile DRAM. *Proc. of 39th Annual International Symposium on Computer Architecture*, pages 37–48, 2012.

[82] D. Meisner, B. T. Gold, and T. F. Wenisch. PowerNap: Eliminating Server Idle Power. *Proc. of 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 205–216, 2009.

[83] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power Management of Online Data-intensive Services. *Proc. of 38th Annual International Symposium on Computer Architecture*, pages 319–330, 2011.

[84] R. Mijat. System Level Benchmarking Analysis of the Cortex-A9 MPCore. *ARM Connected Community Technical Symposium*, 2009.

[85] J. Milthorpe, V. Ganesh, A. Rendell, and D. Grove. X10 as a Parallel Language for Scientific Computation: Practice and Experience. *Proc. of 25th International Parallel & Distributed Processing Symposium*, pages 1080–1088, Anchrage, USA, 2011.

[86] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. *Proc. of 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 261–270, 2009.

[87] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin. Hierarchical Power Management for Asymmetric Multi-Core in Dark Silicon Era. *Proc. of 50th Design Automation Conference*, 2013.

[88] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. *Proc. of 35th Annual International Symposium on Computer Architecture*, pages 63–74, Beijing, China, 2008.

[89] A. K. Nanda, H. Shing, T.-H. Tzen, and L. M. Ni. Resource Contention in Shared-Memory Multiprocessors: A Parameterized Performance Degradation Model. *Journal of Parallel and Distributed Computing*, 12(4):313–328, 1991.

[90] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. *Proc. of 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 208–222, Orlando, USA, 2006.

[91] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual Private Caches. *Proc. of 34th Annual International Symposium on Computer architecture*, pages 57–68, San Diego, USA, 2007.

[92] J. M. Paul and B. H. Meyer. Amdahl's Law Revisited for Single Chip Systems. *International Journal of Parallel Programming*, 35(2):101–123, 2007.

[93] M. Pricopi and T. Mitra. Bahurupi: A polymorphic heterogeneous multi-core architecture. *ACM Transactions on Architure and Code Optimization*, 8(4):22:1–22:21, Jan. 2012.

[94] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[95] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. *Proc. of 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432, Orlando, USA, 2006.

[96] A. Raghavan, L. Emurian, L. Shao, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. Martin. Computational Sprinting on a Hardware/-

Software Testbed. *Proc. of 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 155–166, 2013.

[97] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin. Computational sprinting. *Proc. of 18th IEEE International Symposium on High-Performance Computer Architecture*, pages 1–12, New Orleans, USA, 2012.

[98] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the Bandwidth Wall: Challenges and Avenues for CMP Scaling. *SIGARCH Computer Architure News*, 37(3):371–382, 2009.

[99] J. C. Sancho, D. Kerbyson, and M. Lang. Analyzing the Trade-off between Multiple Memory Controllers and Memory Channels on Multi-core Processor Performance. *Proc. of Workshop on Large-Scale Parallel Processing*, Atlanta, USA, 2010.

[100] D. Sehr and L. V. Kale. Estimating the Inherent Parallelism in Prolog Programs. *Proc of. International Conference on Fifth Generation Computer Systems*, pages 783–790, Tokyo, Japan, 1992.

[101] K. Sevcik. Characterizations of Parallelism in Applications and Their Use in Scheduling. *Proc. of 10th ACM SIGMETRICS Internaltional Conference on Measurement and Modeling of Computer Systems*, pages 171–180, Berkeley, USA, 1989.

[102] S. S. Shende and A. D. Malony. The Tau Parallel Performance System. *International Journal of High Performance Computer Applications*, 20(2):287–311, 2006.

[103] S. Siddha, V. Pallipadi, and A. Mallick. Process Scheduling Challenges in the Era of Multi-core Processors. *Intel Technology Journal*, 11(4), 2007.

[104] K. Singh, M. Bhadauria, and S. A. McKee. Real Time Power Estimation and Thread Scheduling via Performance Counters. *SIGARCH Computer Architure News*, 37:46–55, 2009.

[105] K. Singh, M. Curtis-Maury, S. McKee, F. Blagojevic, D. Nikolopoulos, B. de Supinski, and M. Schulz. Comparing Scalability Prediction Strategies on an SMP of CMPs. *Proc. of 16th International European Conference on Parallel and Distributed Computing*, Ischia, Italy, 2010.

[106] Y. Solihin, V. Lam, and J. Torrellas. Scal-Tool: Pinpointing and Quantifying Scalability Bottlenecks in DSM Multiprocessors. *Proc. of 11th ACM/IEEE Conference on Supercomputing*, Portland, USA, 1999.

[107] C. Stewart, T. Kelly, and A. Zhang. Exploiting Nonstationarity for Performance Prediction. *ACM SIGOPS Operating Systems Review*, 41(3):31–44, 2007.

[108] V. Suhendra and T. Mitra. Exploring Locking & Partitioning for Predictable Shared Caches on Multi-cores. *Proc. of 45th Annual Design Automation Conference*, pages 300–303, 2008.

[109] J. Tao, W. Karl, and M. Schulz. Memory Access Behavior Analysis of NUMA-based Shared Memory Programs. *Scientific Programming*, 10(1):45–53, Jan. 2002.

[110] Y. C. Tay. *Analytical Performance Modeling for Computer Systems*. Synthesis Lectures on Computer Science. Morgan & Claypool Publishers, 2010.

[111] E. Thereska, B. Doebel, A. X. Zheng, and P. Nobel. Practical Performance Models for Complex, Popular Applications. *Proc of. 31st SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, New York, USA, 2010.

[112] Travis Lanier. *Exploring the design of the Cortex-A15 processor*, 2011. ARM Technical Reports.

[113] H.-L. Truong and T. Fahringer. SCALEA: A Performance Analysis Tool for Parallel Programs. *Concurrency and Computation: Practice and Experience Journal*, 15(11-12):1001–1025, 2003.

[114] B. M. Tudor and Y. M. Teo. A Practical Approach for Performance Analysis of Shared-Memory Programs. *Proc. of 25th IEEE International Parallel & Distributed Processing Symposium*, pages 652–663, Anchorage, USA, 2011.

[115] B. M. Tudor and Y. M. Teo. Towards Modelling Parallelism and Energy Performance of Multicore Systems. *Proc. of 26th IEEE International Parallel & Distributed Processing Symposium PhD Forum*, Shanghai, China, 2012.

[116] B. M. Tudor and Y. M. Teo. On Understanding the Energy Consumption of ARM-based Multicore Servers. *Proc. of 34th ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 267–278, Pittsburgh, USA, 2013.

[117] B. M. Tudor, Y. M. Teo, and S. See. Understanding Off-Chip Memory Contention of Parallel Programs in Multicore Systems. *Proc. of 40th International Conference on Parallel Processing*, pages 602–611, Taipei, Taiwan, 2011.

[118] S. J. E. Wilton and N. P. Jouppi. CACTI: An Enhanced Cache Access and Cycle Time Model. *IEEE Journal of Solid-State Circuits*, 31:677–688, 1996.

[119] F. Wolf and B. Mohr. KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Applications. *Proc. of 9th European Conference on Parallel Computing*, pages 1301–1304, Klagenfurt, Austria, 2003.

[120] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Architure News*, 23(1):20–24, 1995.

[121] Y. Xie and G. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. *Proc. of 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects*, Beijing, China, 2008.

[122] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin. The Design and Use of SimplePower: a Cycle-Accurate Energy Estimation Tool. *Proc. of 37th Annual Design Automation Conference*, pages 340–345, Los Angeles, USA, 2000.

[123] D. H. Yoon, J. Chang, N. Muralimanohar, and P. Ranganathan. BOOM: Enabling Mobile Memory Based Low-power Server DIMMs. *Proc. of 39th Annual International Symposium on Computer Architecture*, pages 25–36, Portland, Oregon, 2012.

[124] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. *Proc. of 15th Edition of Architectural Support for Programming Languages and Operating Systems*, pages 129–142, Pittsburgh, USA, 2010.

Appendix A

Validation Results

A.1 Validation of Memory Contention Model

A.1.1 Intel UMA

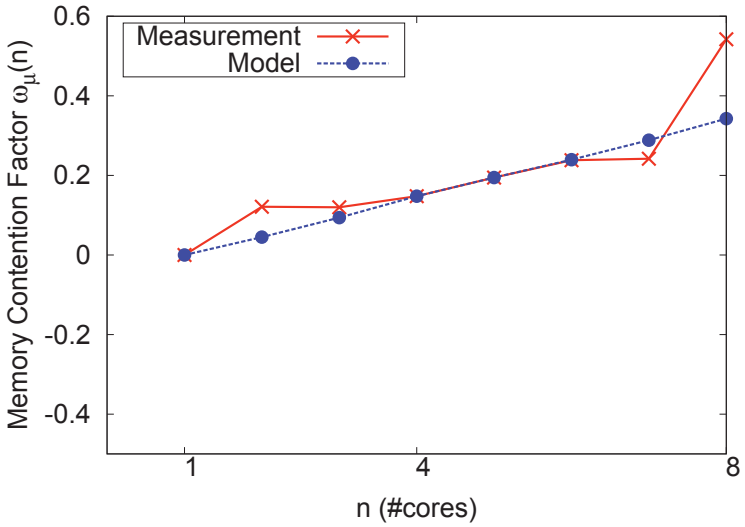


Figure A.1: Memory contention validation: *BT.W* on Intel UMA

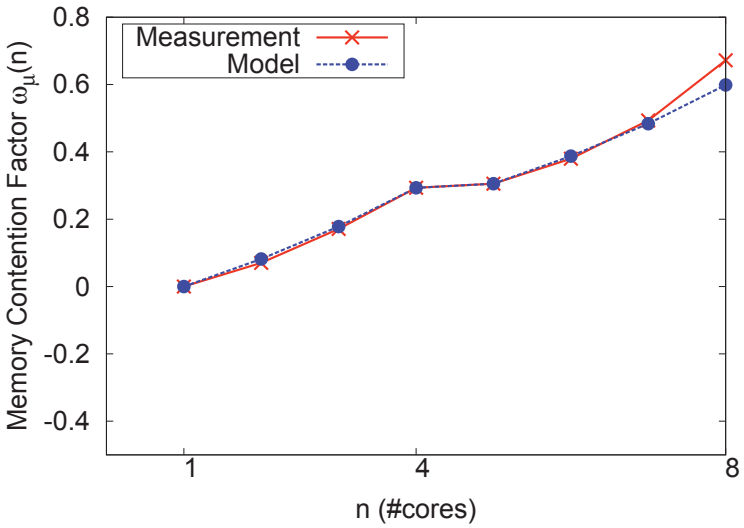


Figure A.2: Memory contention validation: *BT.A* on Intel UMA

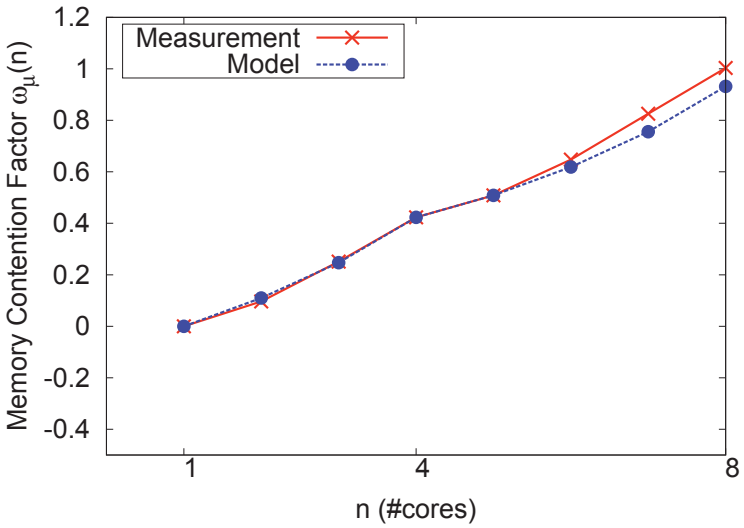


Figure A.3: Memory contention validation: *BT.B* on Intel UMA

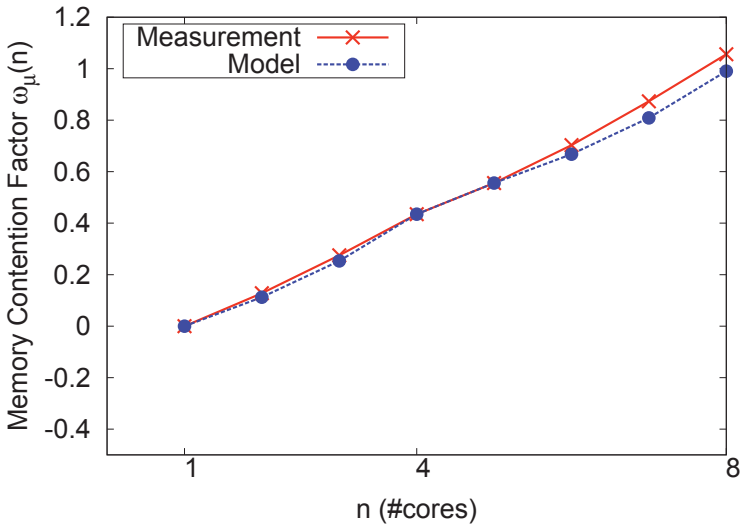


Figure A.4: Memory contention validation: *BT.C* on Intel UMA

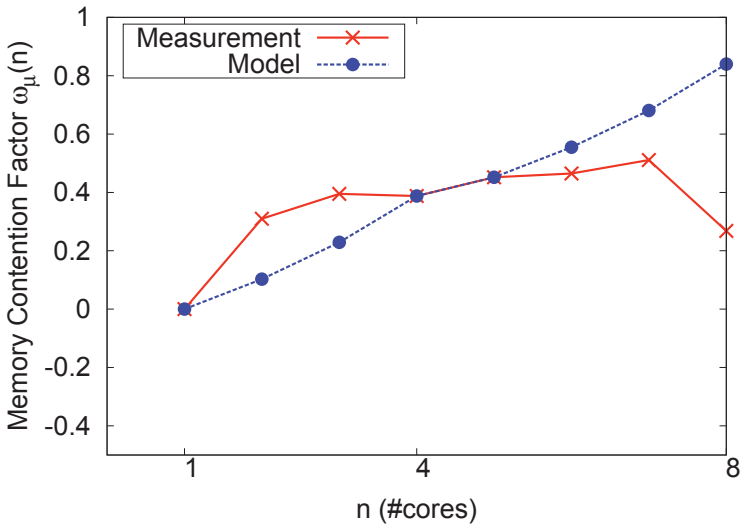


Figure A.5: Memory contention validation: *CG.W* on Intel UMA

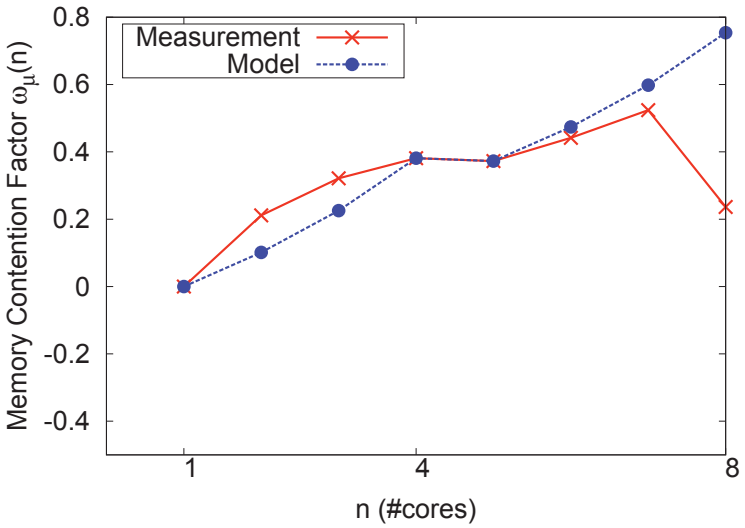


Figure A.6: Memory contention validation: *CG.A* on Intel UMA

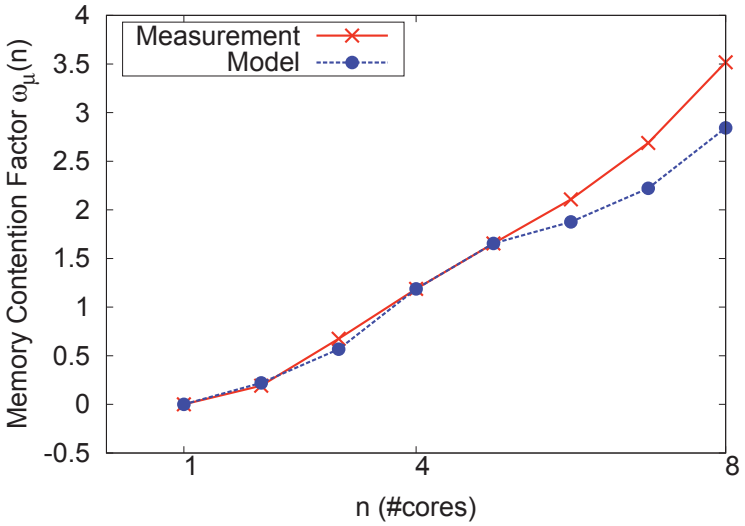


Figure A.7: Memory contention validation: *CG.B* on Intel UMA

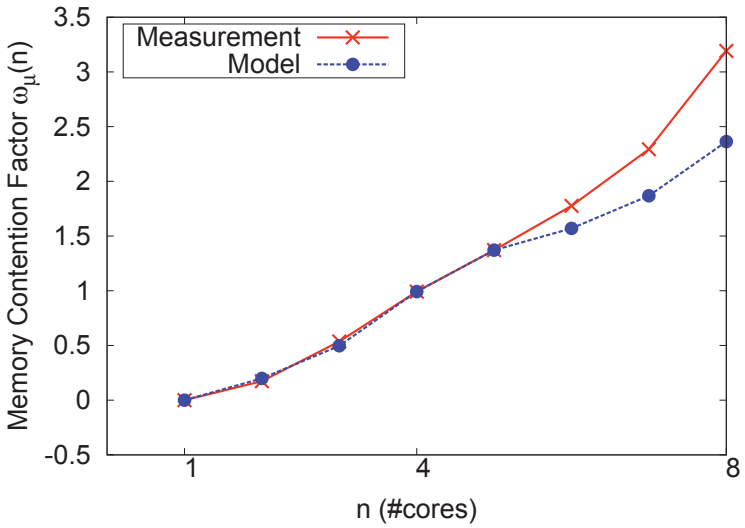


Figure A.8: Memory contention validation: *CG.C* on Intel UMA

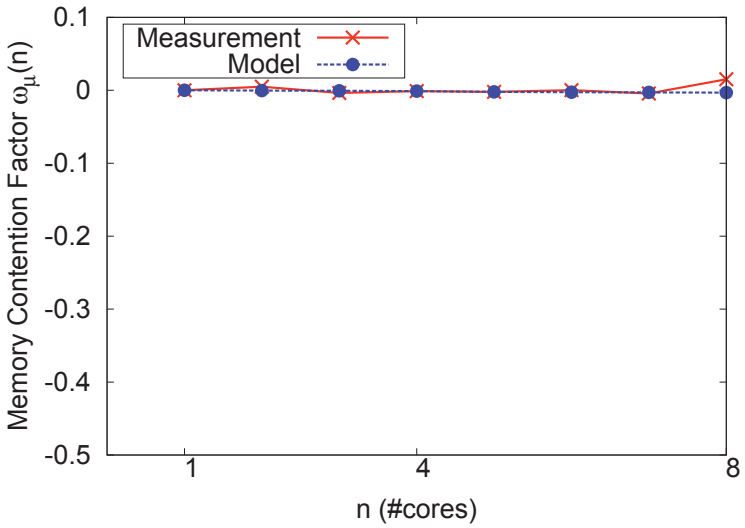


Figure A.9: Memory contention validation: *EP.W* on Intel UMA

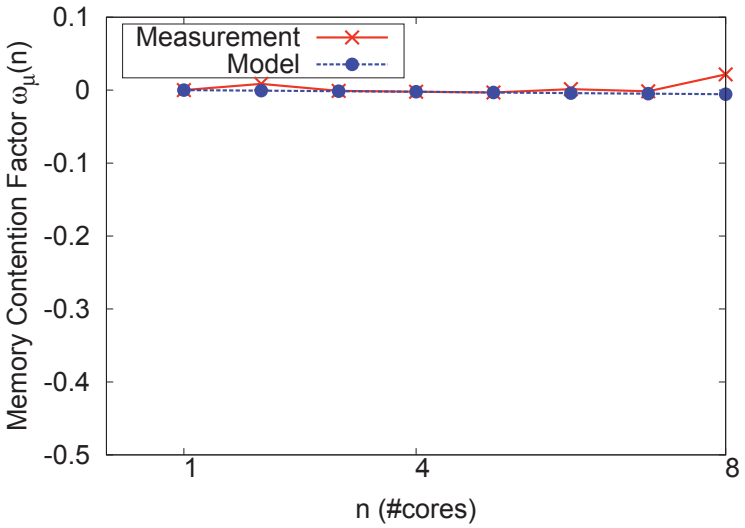


Figure A.10: Memory contention validation: *EP.A* on Intel UMA

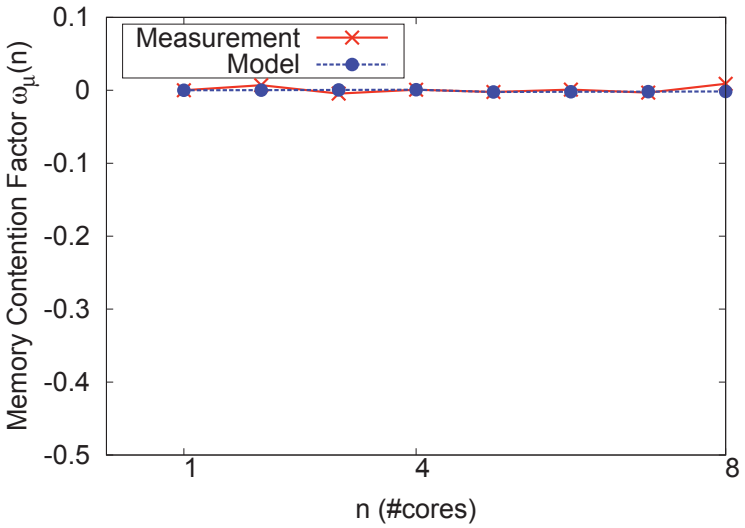


Figure A.11: Memory contention validation: *EP.B* on Intel UMA

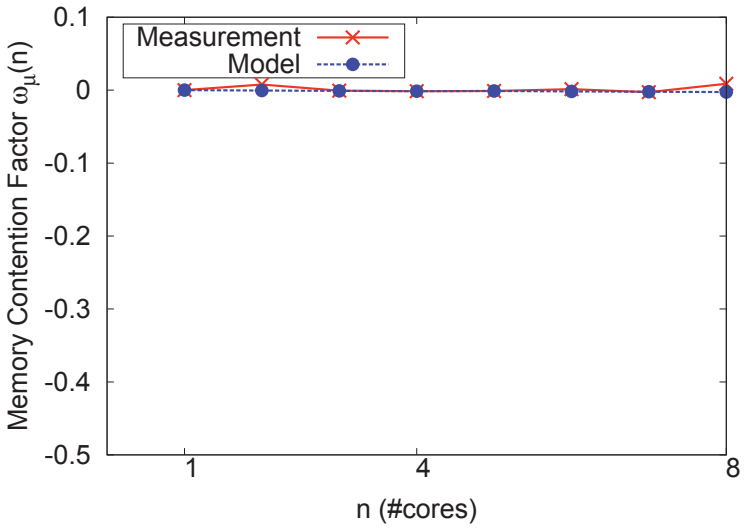


Figure A.12: Memory contention validation: *EP.C* on Intel UMA

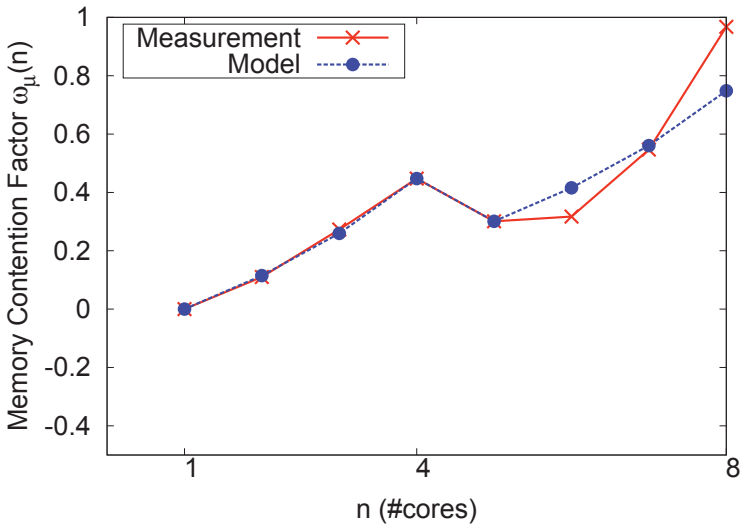


Figure A.13: Memory contention validation: *FT.W* on Intel UMA

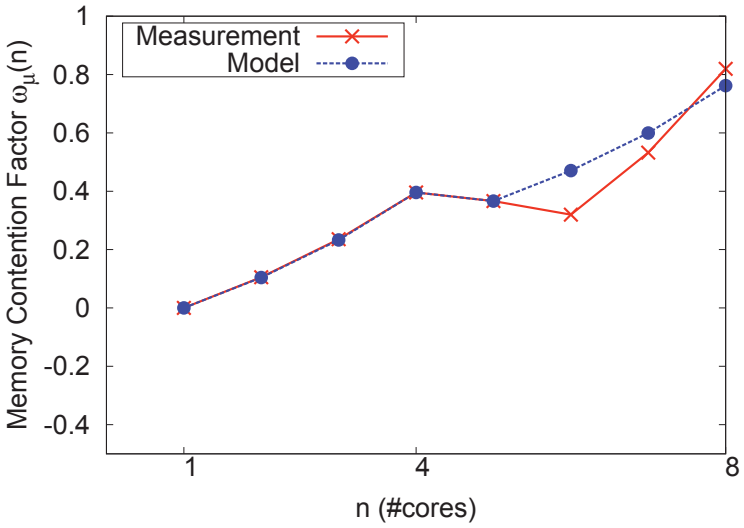


Figure A.14: Memory contention validation: $FT.A$ on Intel UMA

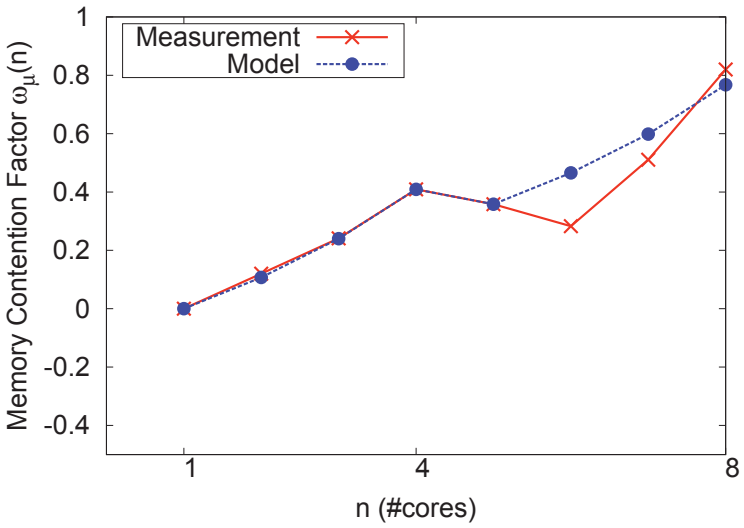


Figure A.15: Memory contention validation: $FT.B$ on Intel UMA

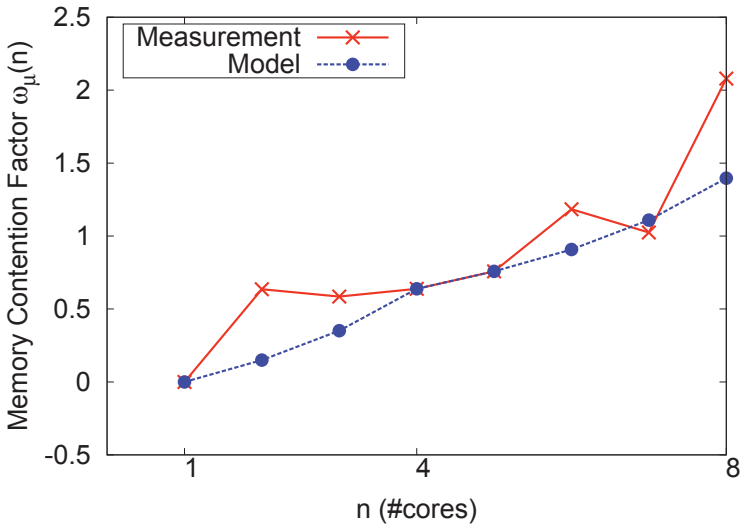


Figure A.16: Memory contention validation: $IS.W$ on Intel UMA

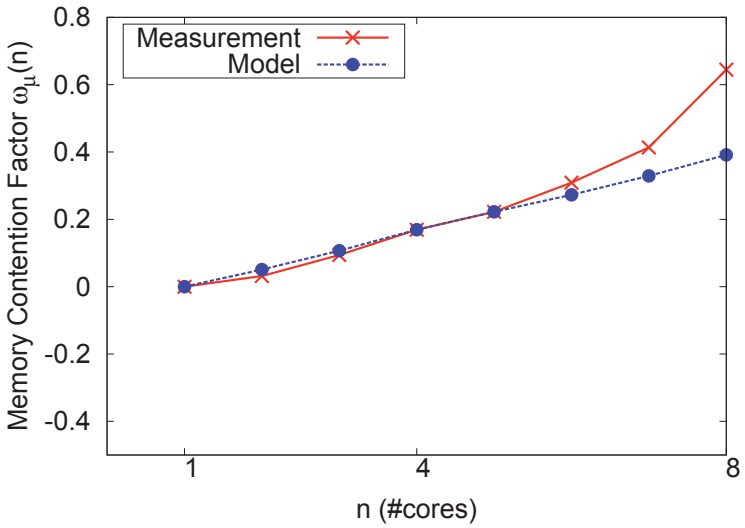


Figure A.17: Memory contention validation: $IS.A$ on Intel UMA

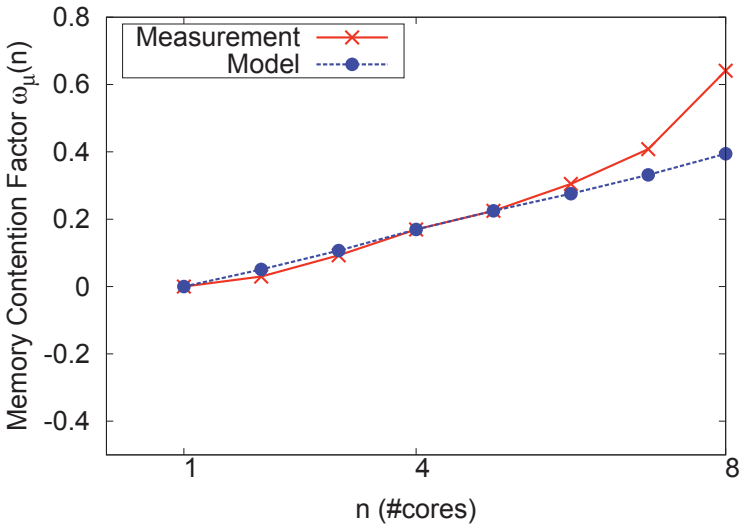


Figure A.18: Memory contention validation: $IS.B$ on Intel UMA

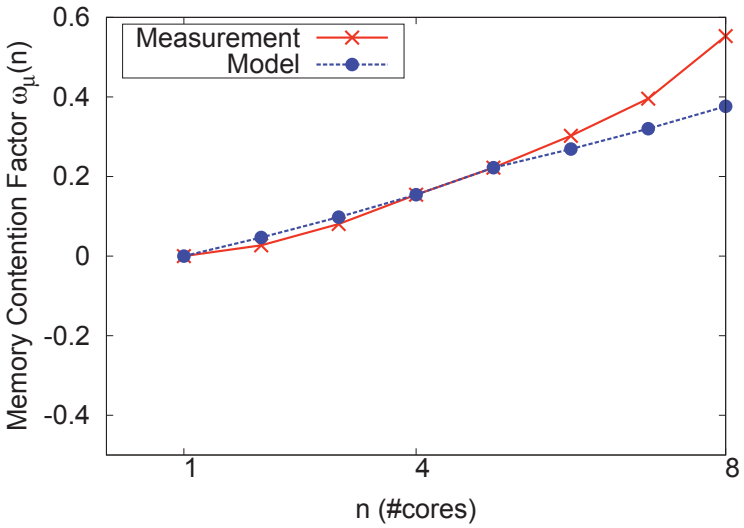


Figure A.19: Memory contention validation: $IS.C$ on Intel UMA

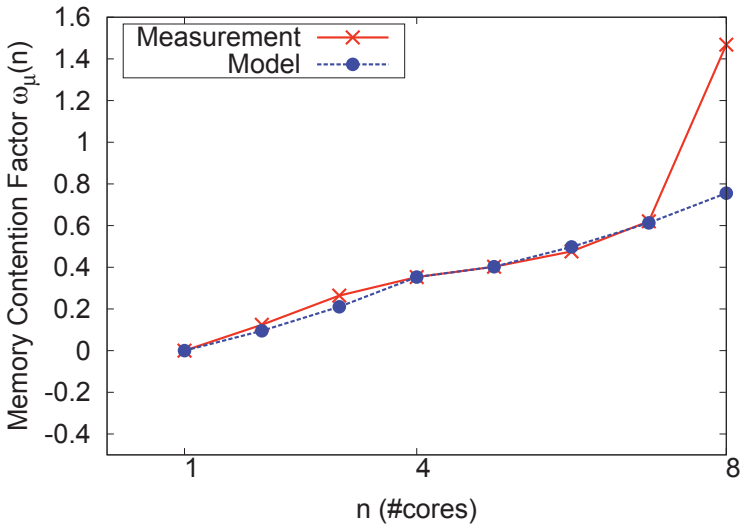


Figure A.20: Memory contention validation: *SP.W* on Intel UMA

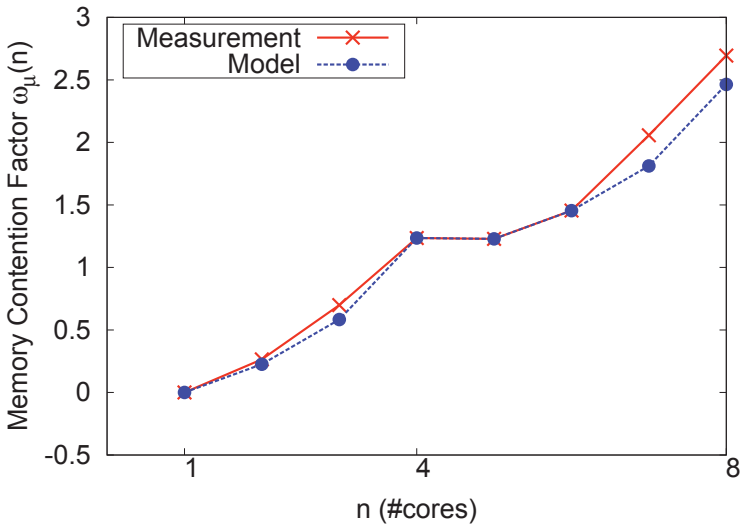


Figure A.21: Memory contention validation: *SP.A* on Intel UMA

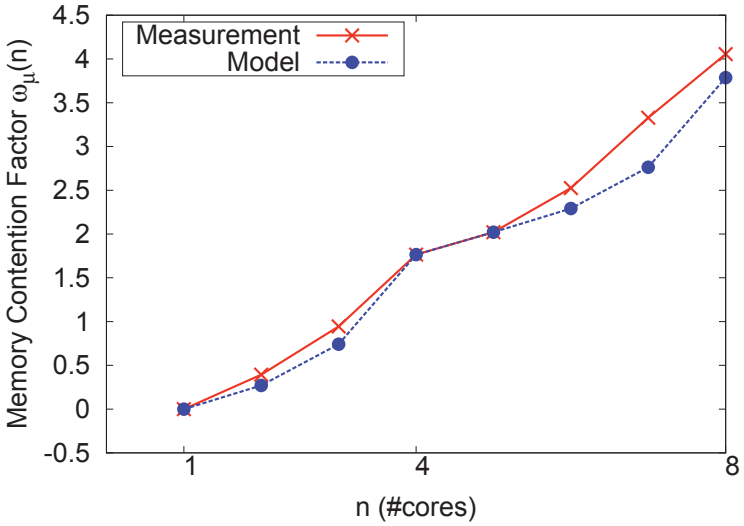


Figure A.22: Memory contention validation: *SP.B* on Intel UMA

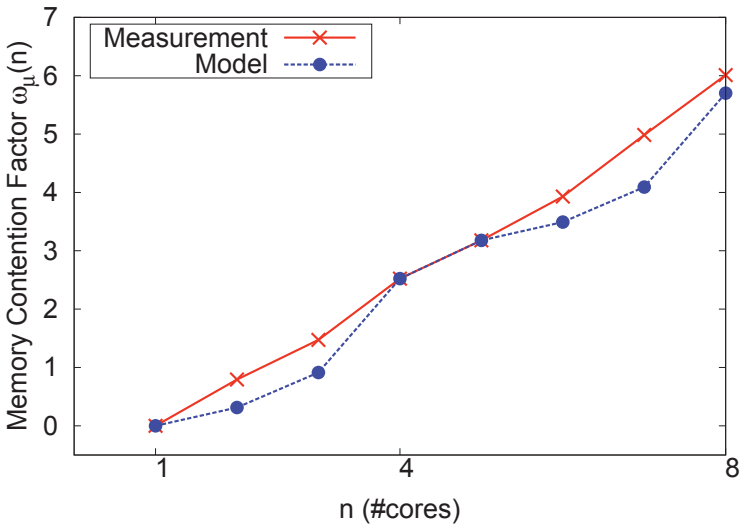


Figure A.23: Memory contention validation: *SP.C* on Intel UMA

A.1.2 Intel NUMA 2

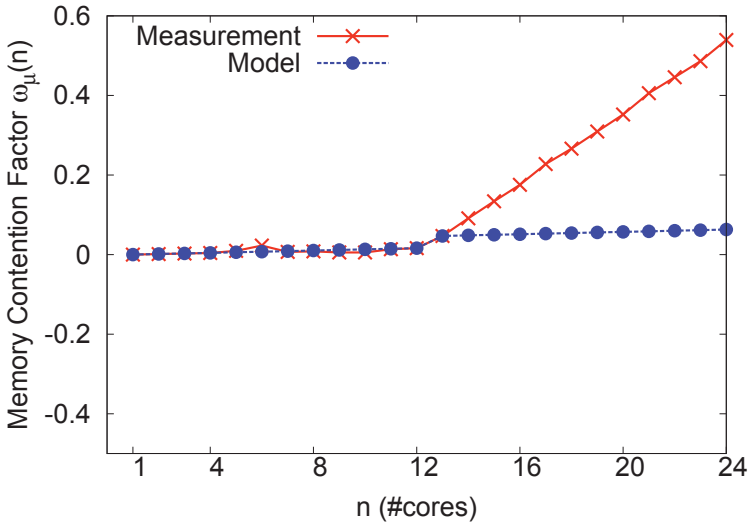


Figure A.24: Memory contention validation: *EP.C* on Intel NUMA

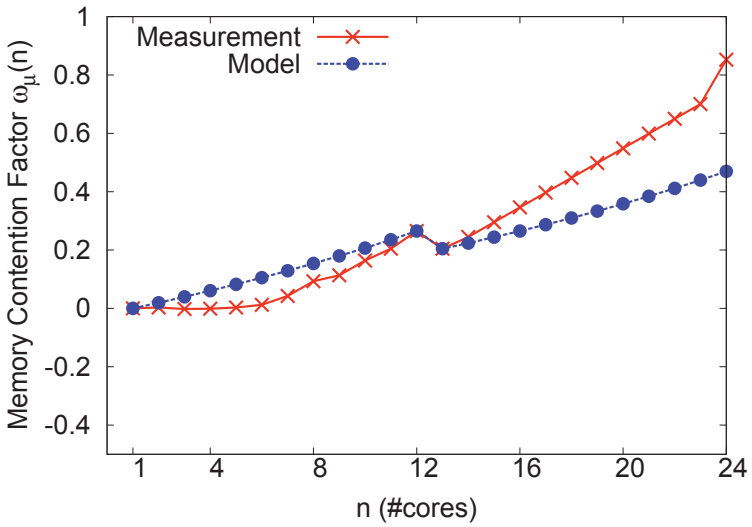


Figure A.25: Memory contention validation: *IS.C* on Intel NUMA

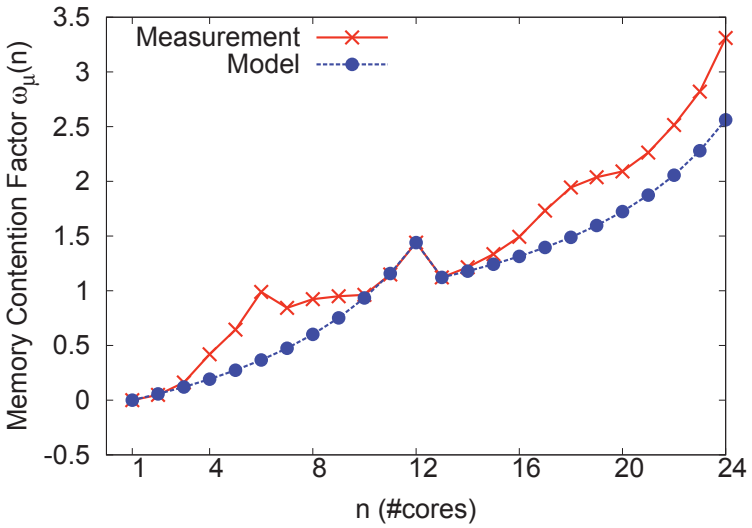


Figure A.26: Memory contention validation: *CG.C* on Intel NUMA

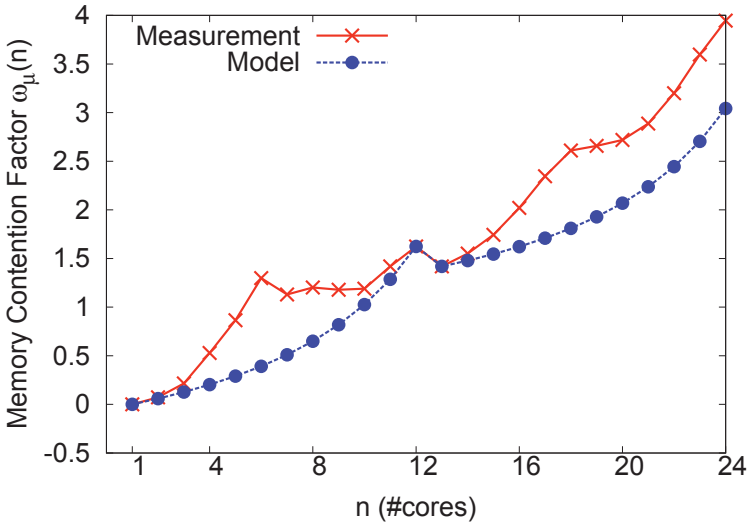


Figure A.27: Memory contention validation: *FT.C* on Intel NUMA

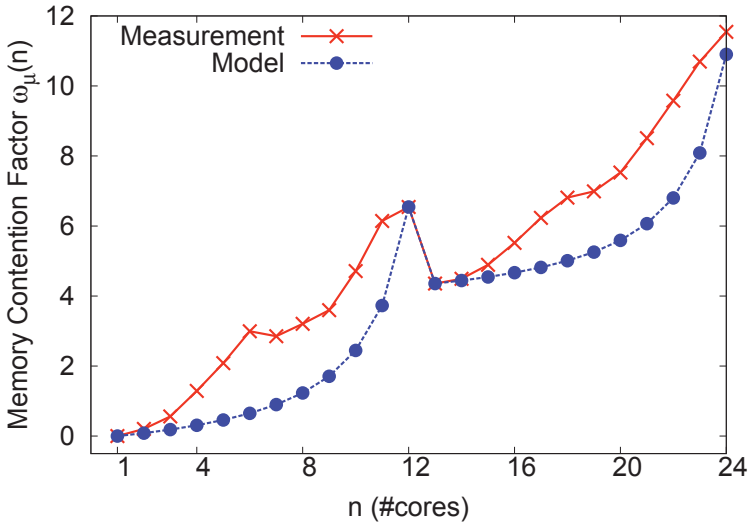


Figure A.28: Memory contention validation: *SP.C* on Intel NUMA

A.1.3 AMD NUMA

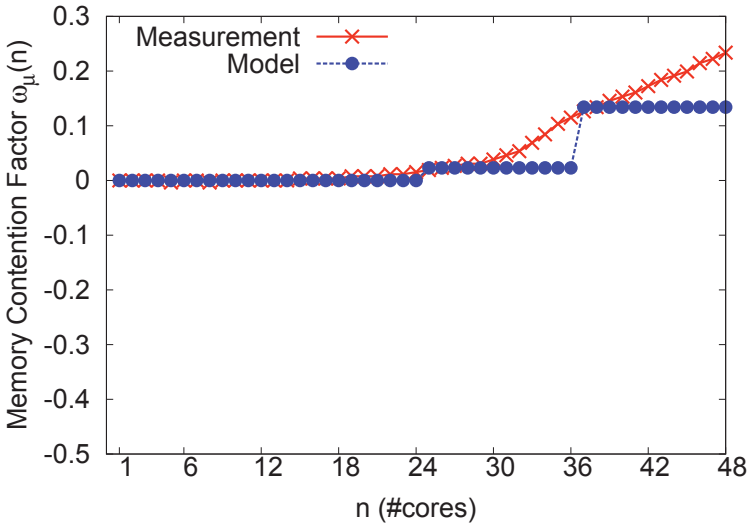


Figure A.29: Memory contention validation: *EP.C* on AMD NUMA

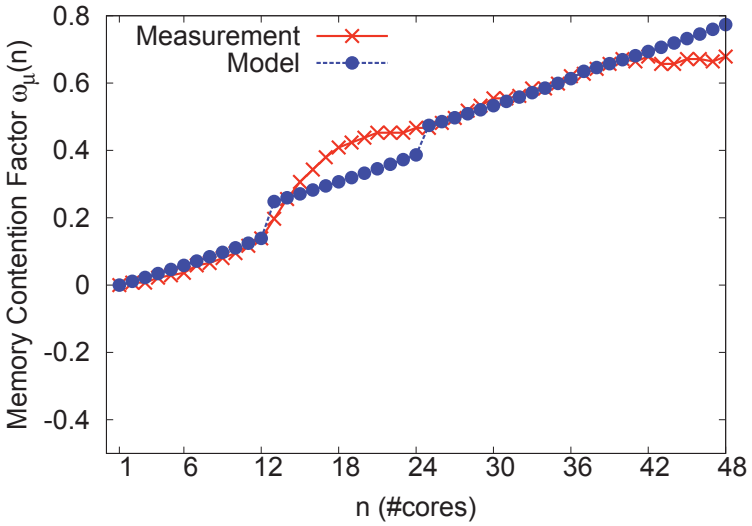


Figure A.30: Memory contention validation: *IS.C* on AMD NUMA

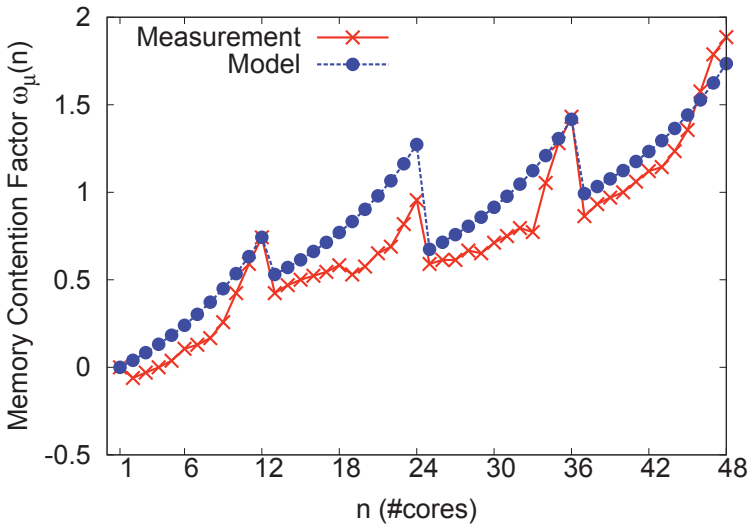


Figure A.31: Memory contention validation: *CG.C* on AMD NUMA

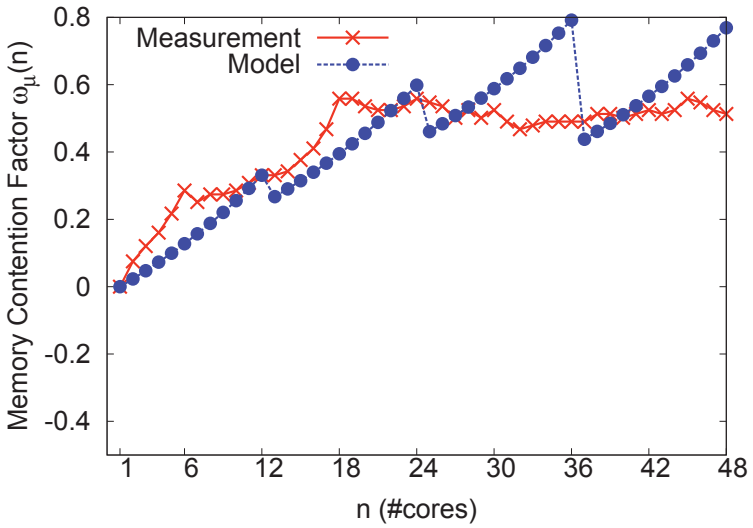


Figure A.32: Memory contention validation: *FT.C* on AMD NUMA

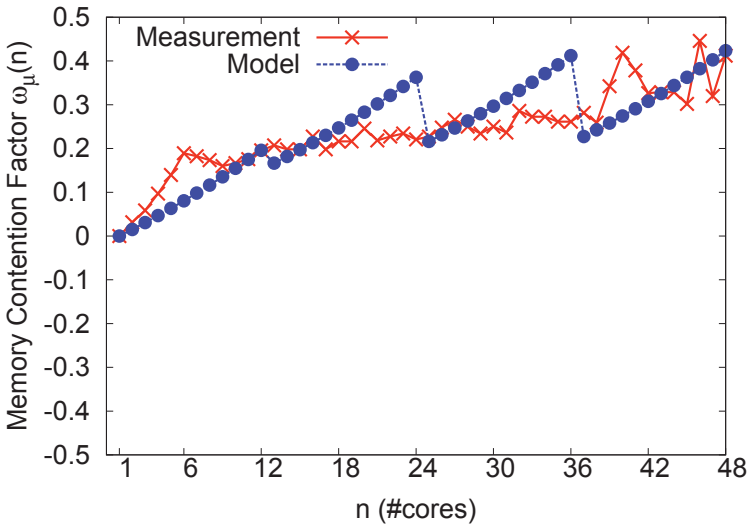


Figure A.33: Memory contention validation: *BT.C* on AMD NUMA

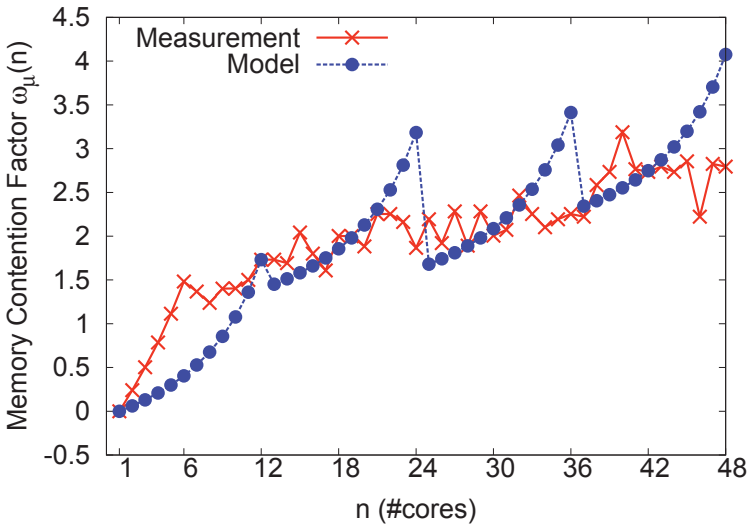


Figure A.34: Memory contention validation: *SP.C* on AMD NUMA

A.2 Validation of Parallelism and Energy Model on ARM Cortex-A9

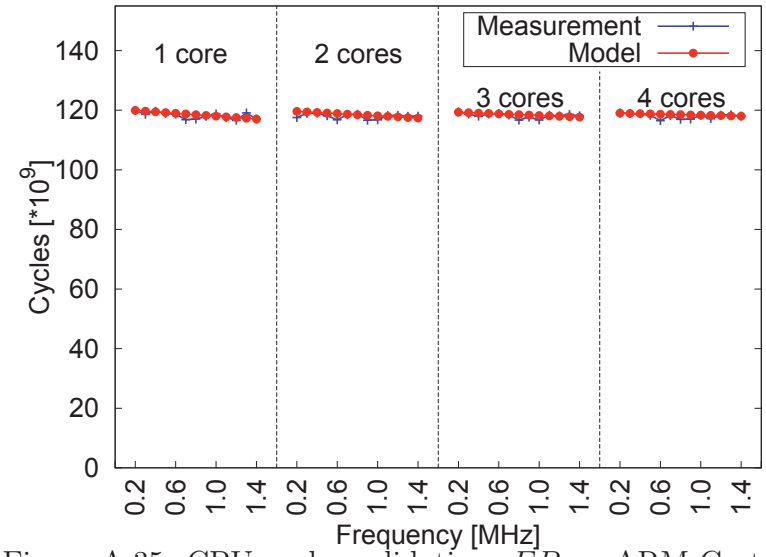


Figure A.35: CPU cycles validation: *EP* on ARM Cortex-A9

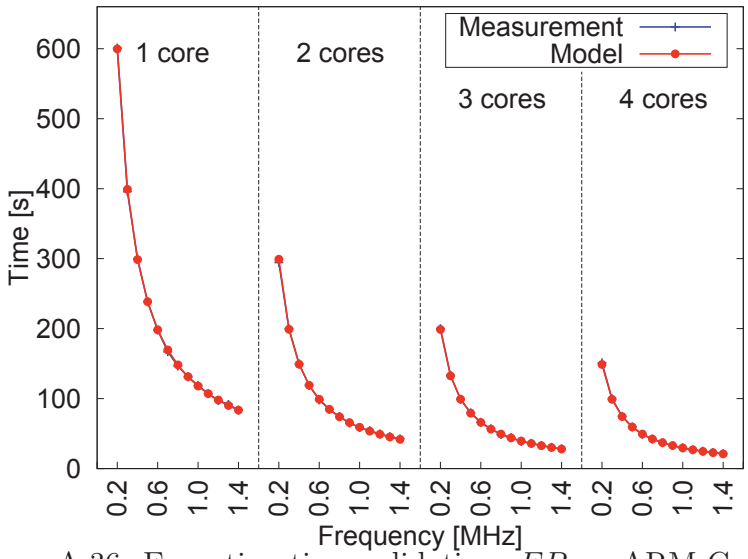


Figure A.36: Execution time validation: *EP* on ARM Cortex-A9

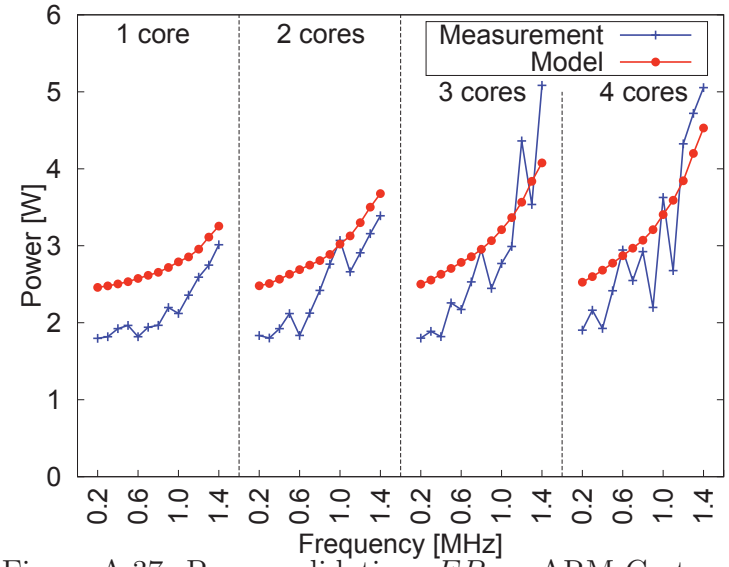


Figure A.37: Power validation: EP on ARM Cortex-A9

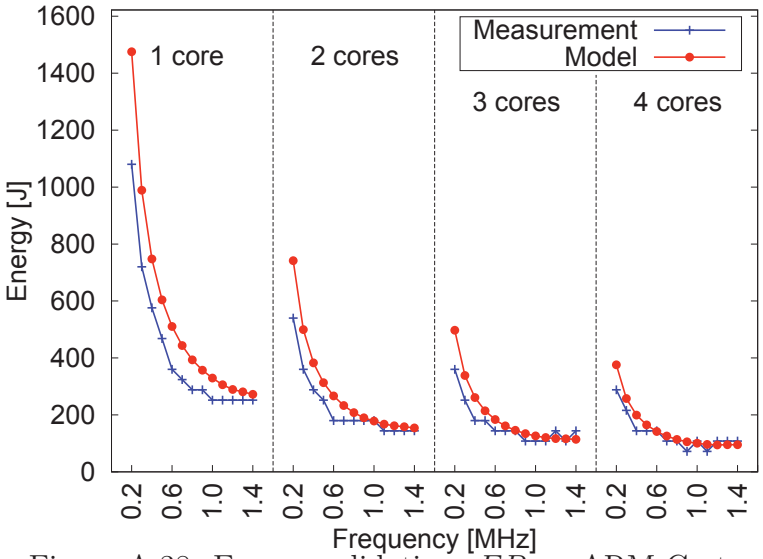


Figure A.38: Energy validation: EP on ARM Cortex-A9

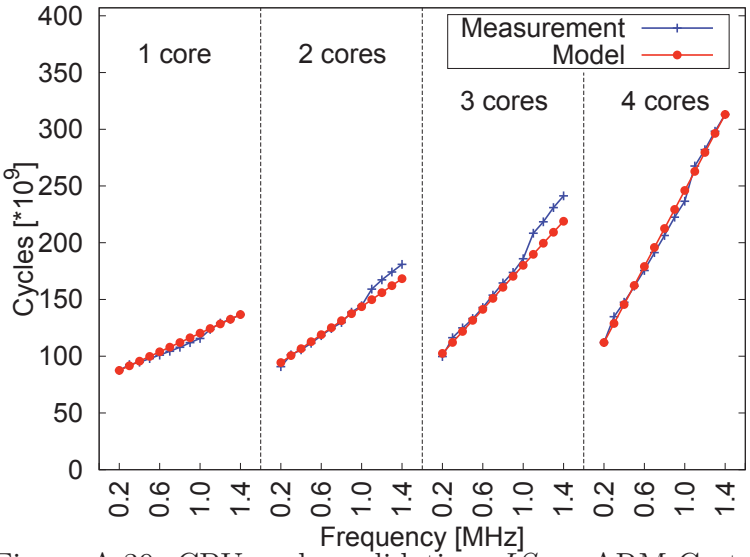


Figure A.39: CPU cycles validation: *IS* on ARM Cortex-A9

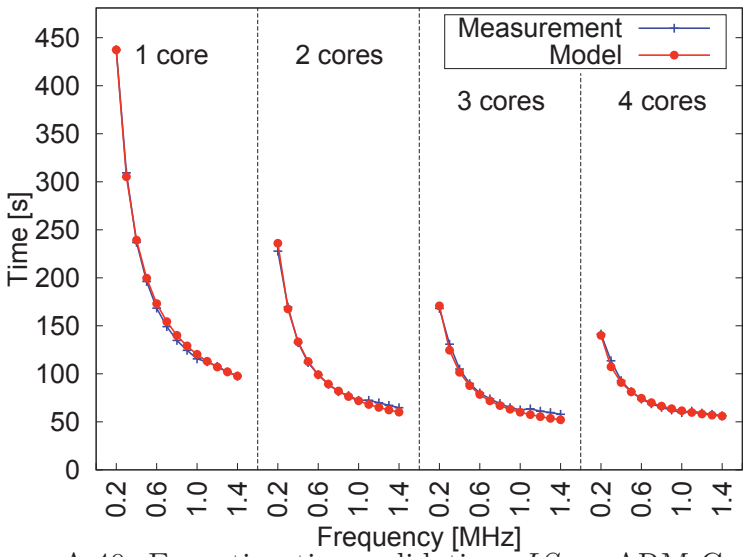


Figure A.40: Execution time validation: *IS* on ARM Cortex-A9

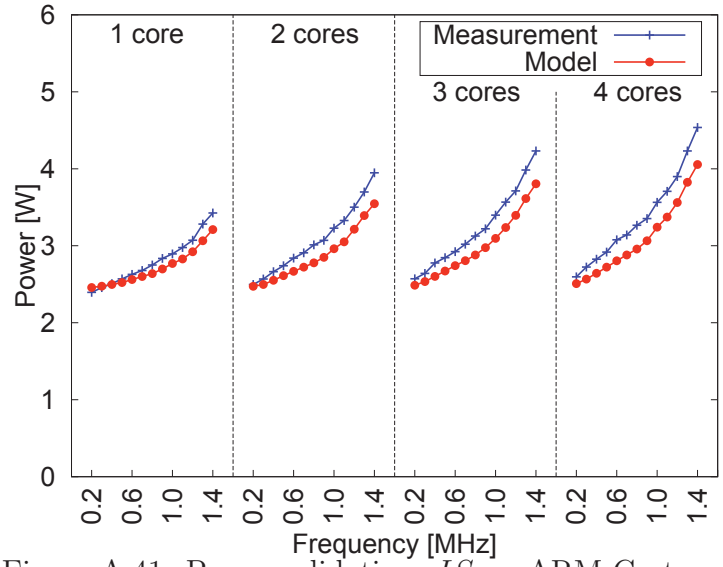


Figure A.41: Power validation: *IS* on ARM Cortex-A9

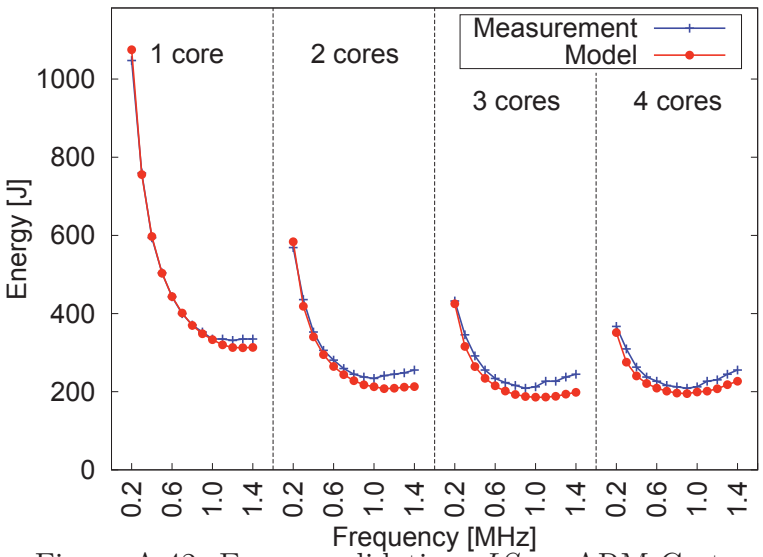


Figure A.42: Energy validation: *IS* on ARM Cortex-A9

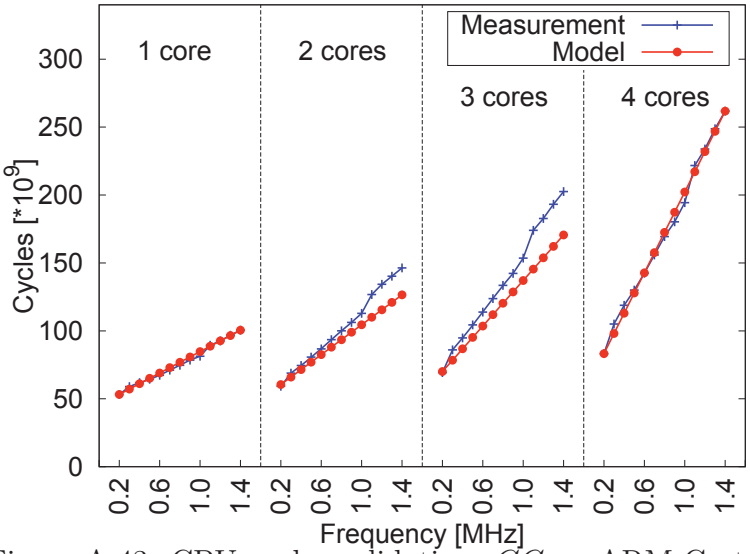


Figure A.43: CPU cycles validation: *CG* on ARM Cortex-A9

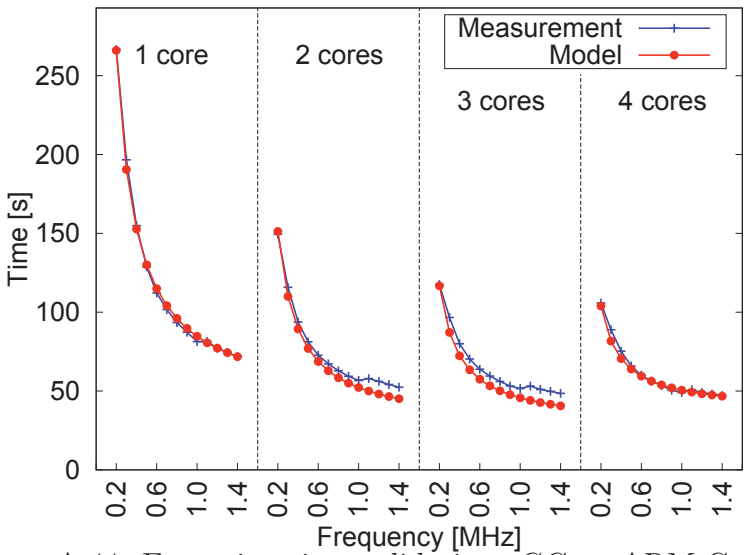


Figure A.44: Execution time validation: *CG* on ARM Cortex-A9

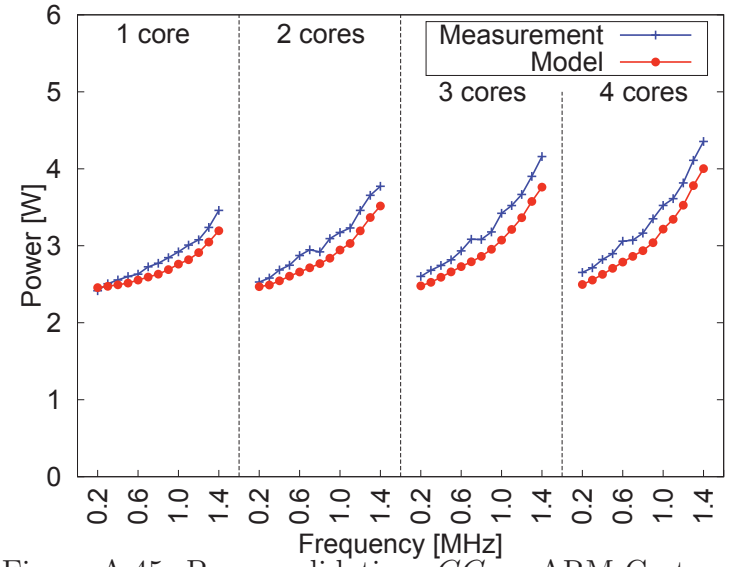


Figure A.45: Power validation: *CG* on ARM Cortex-A9

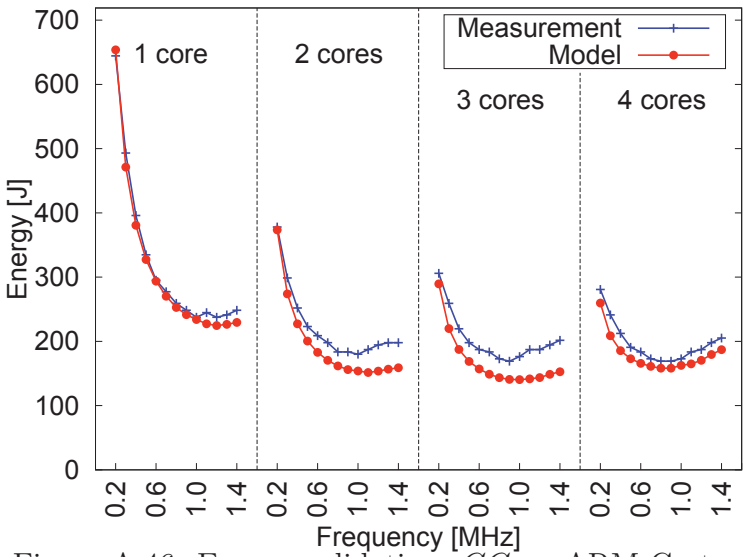


Figure A.46: Energy validation: *CG* on ARM Cortex-A9

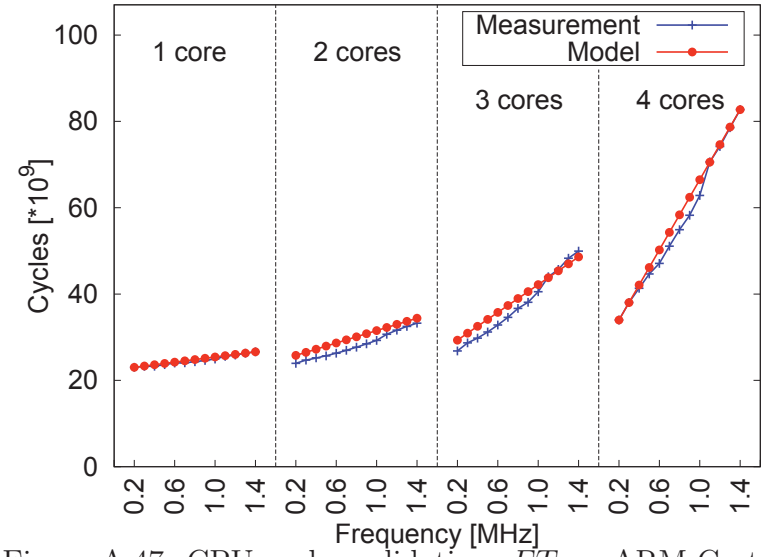


Figure A.47: CPU cycles validation: *FT* on ARM Cortex-A9

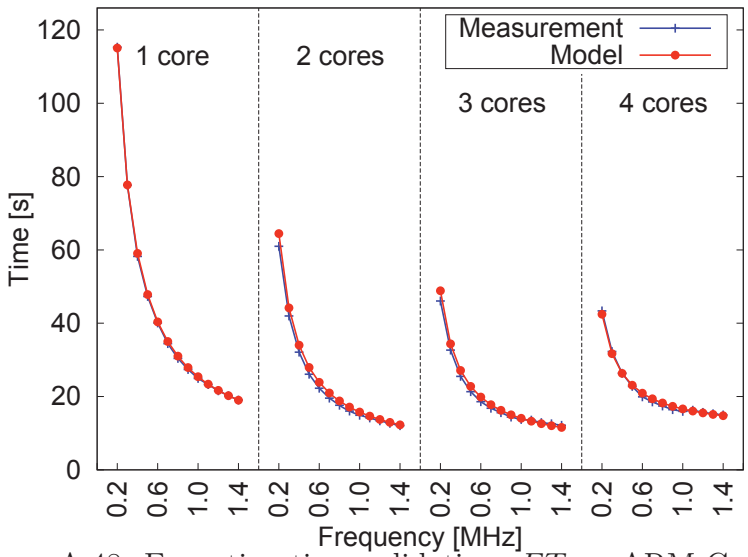


Figure A.48: Execution time validation: *FT* on ARM Cortex-A9

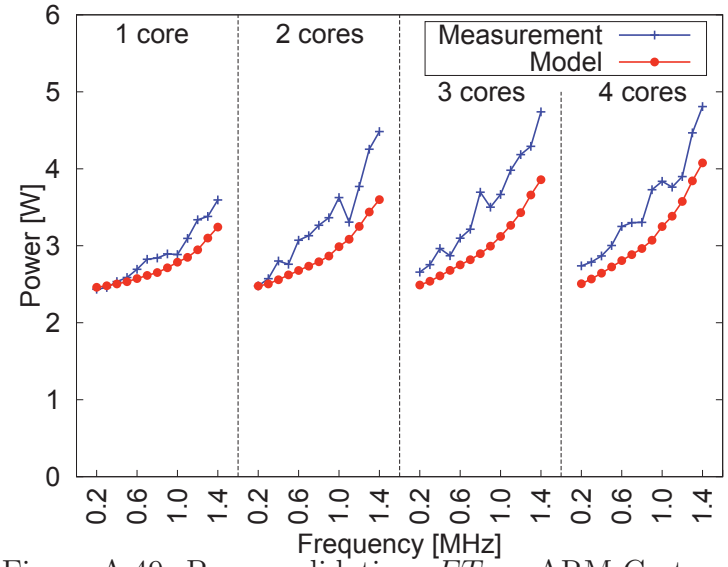


Figure A.49: Power validation: *FT* on ARM Cortex-A9

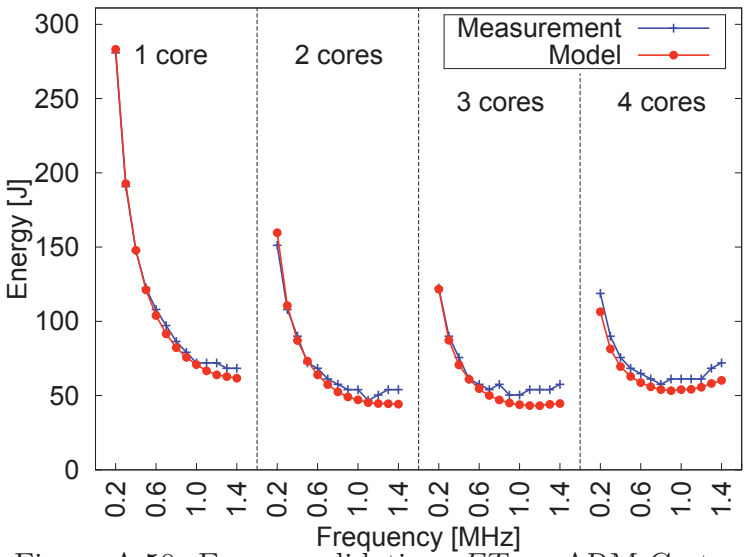


Figure A.50: Energy validation: *FT* on ARM Cortex-A9

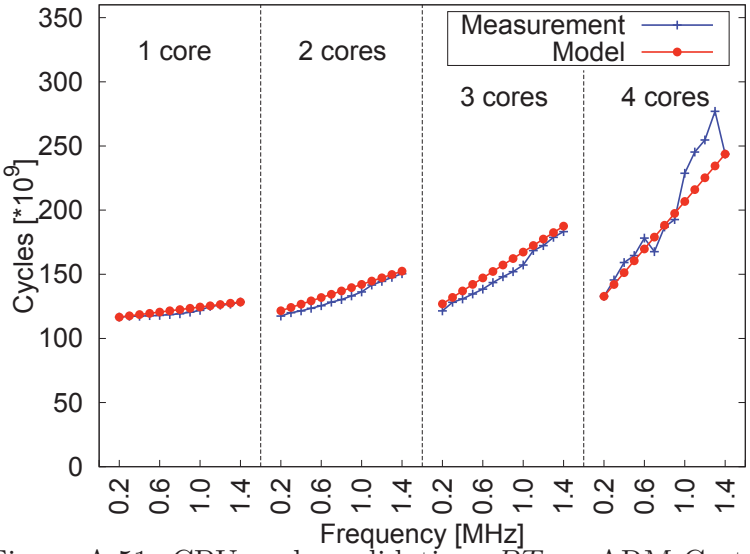


Figure A.51: CPU cycles validation: *BT* on ARM Cortex-A9

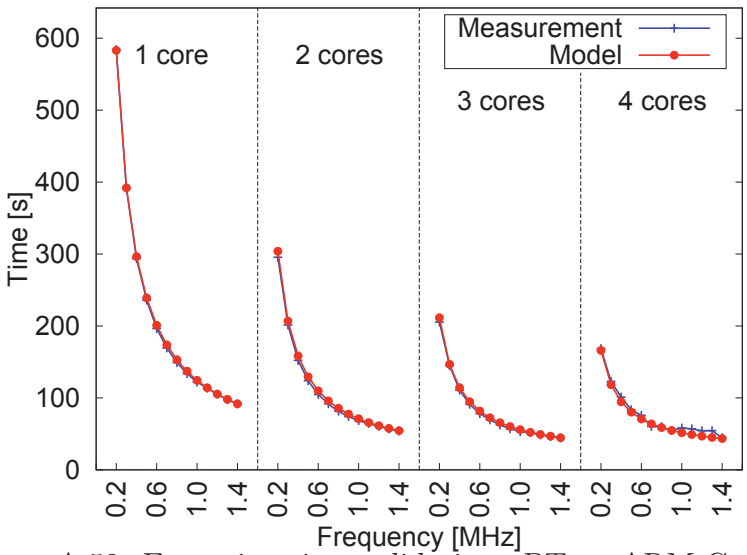


Figure A.52: Execution time validation: *BT* on ARM Cortex-A9

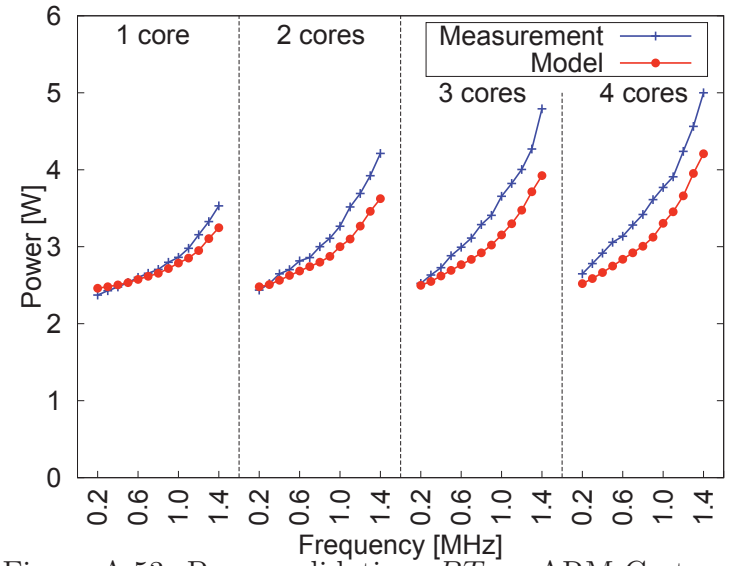


Figure A.53: Power validation: *BT* on ARM Cortex-A9

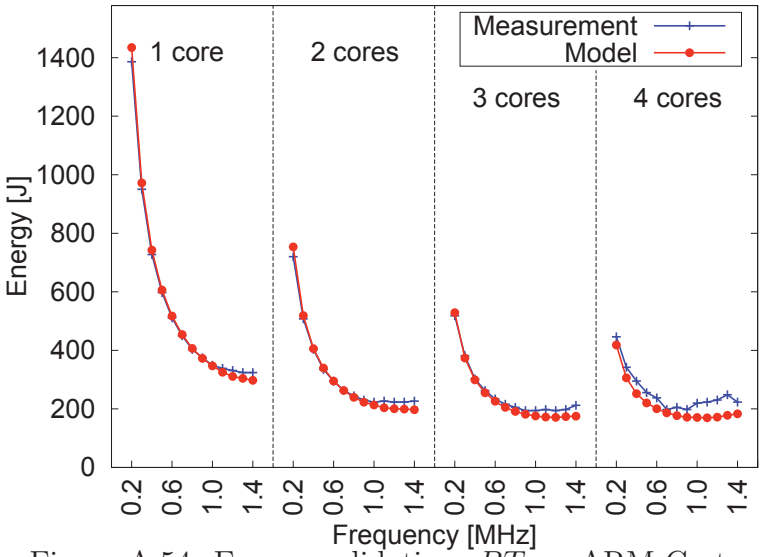


Figure A.54: Energy validation: *BT* on ARM Cortex-A9

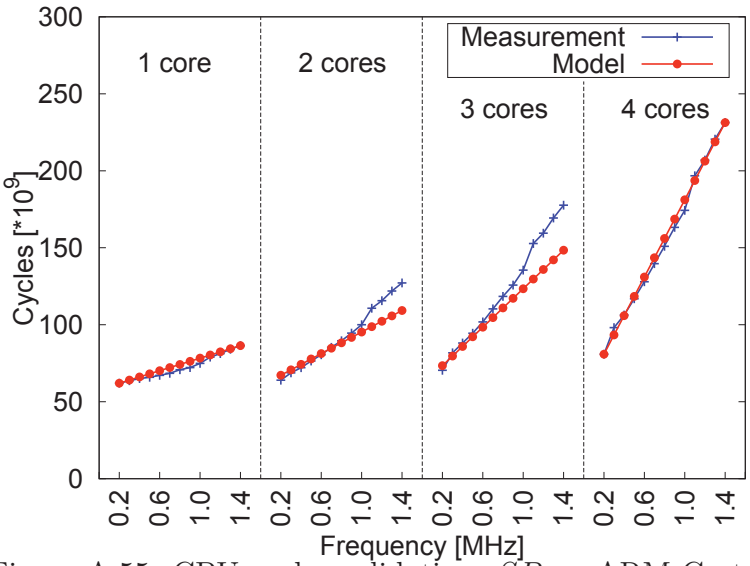


Figure A.55: CPU cycles validation: *SP* on ARM Cortex-A9

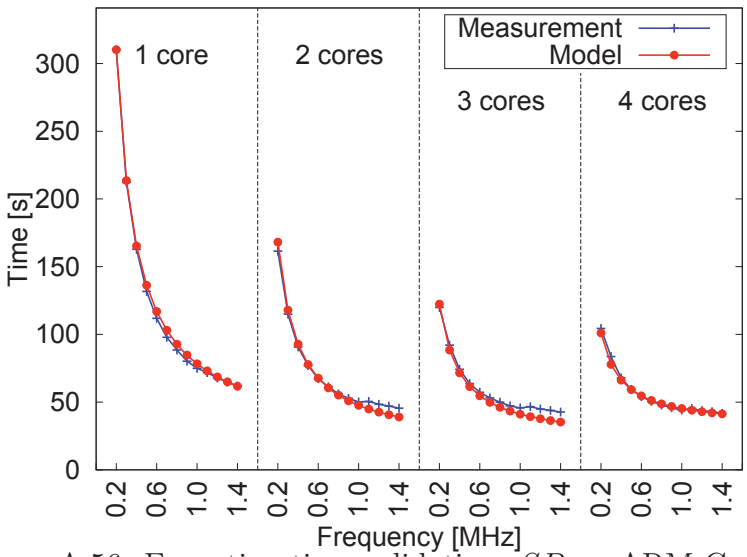


Figure A.56: Execution time validation: *SP* on ARM Cortex-A9

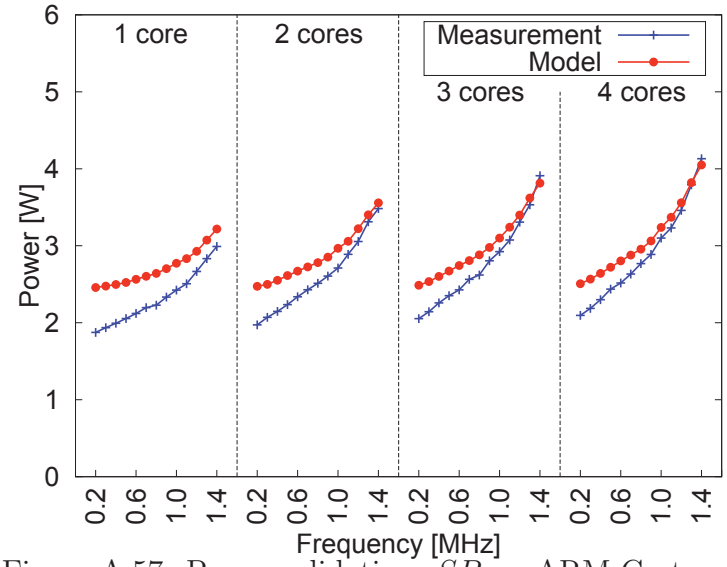


Figure A.57: Power validation: SP on ARM Cortex-A9

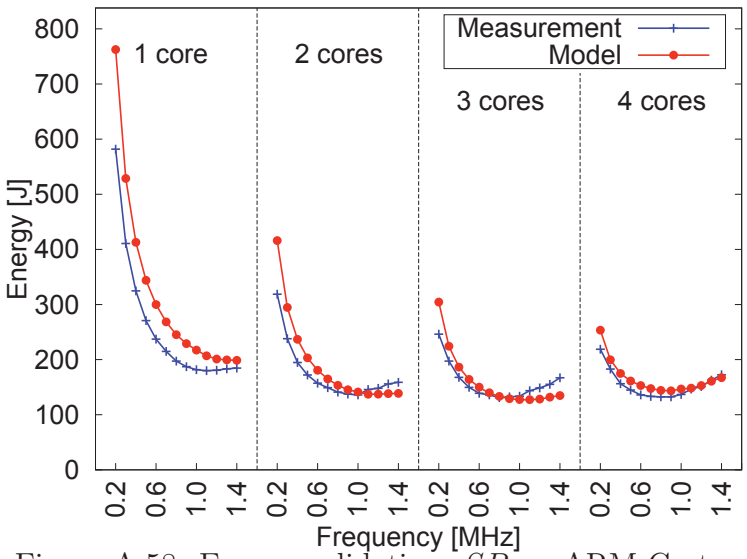


Figure A.58: Energy validation: SP on ARM Cortex-A9

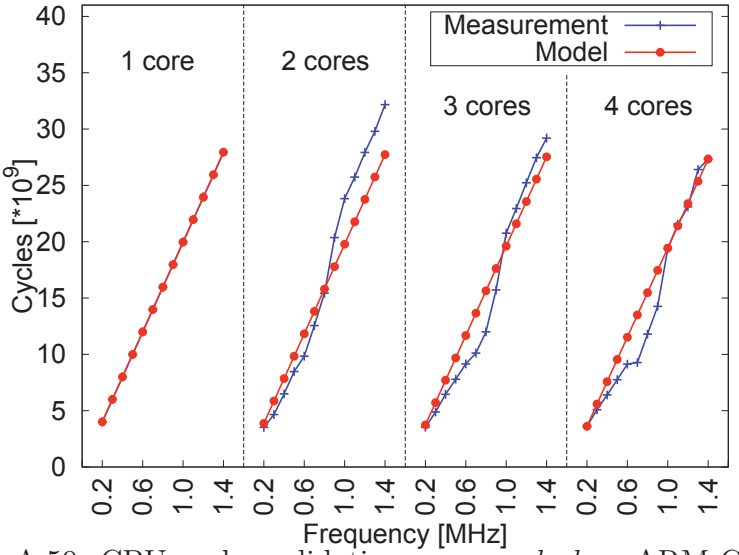


Figure A.59: CPU cycles validation: *memcached* on ARM Cortex-A9

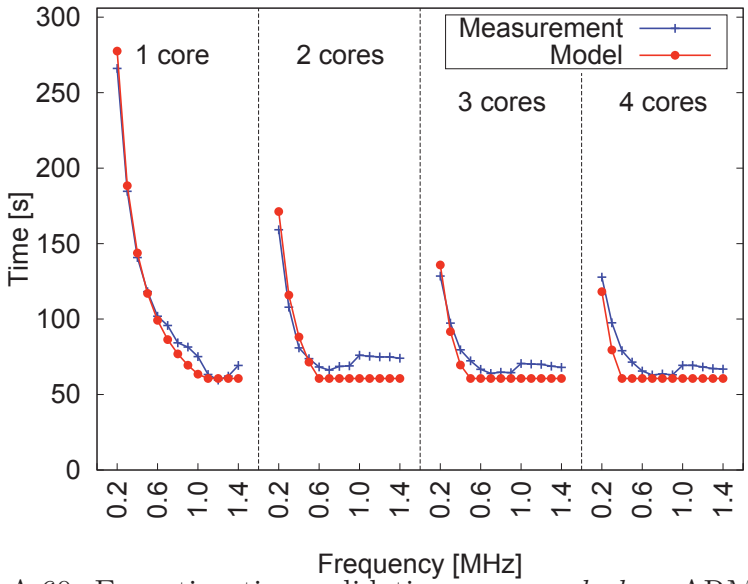


Figure A.60: Execution time validation: *memcached* on ARM Cortex-A9

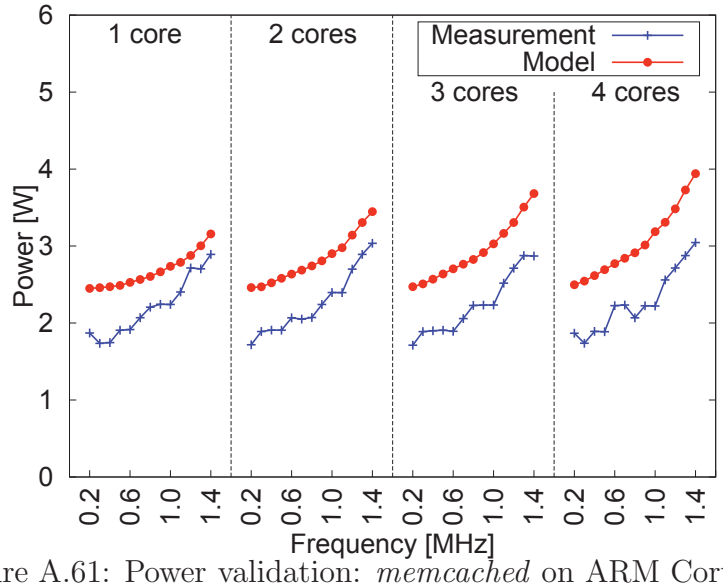


Figure A.61: Power validation: *memcached* on ARM Cortex-A9

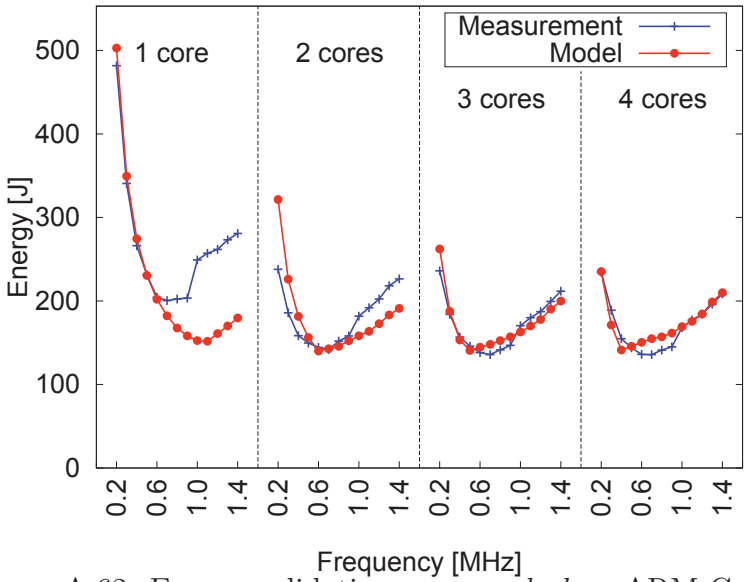


Figure A.62: Energy validation: *memcached* on ARM Cortex-A9