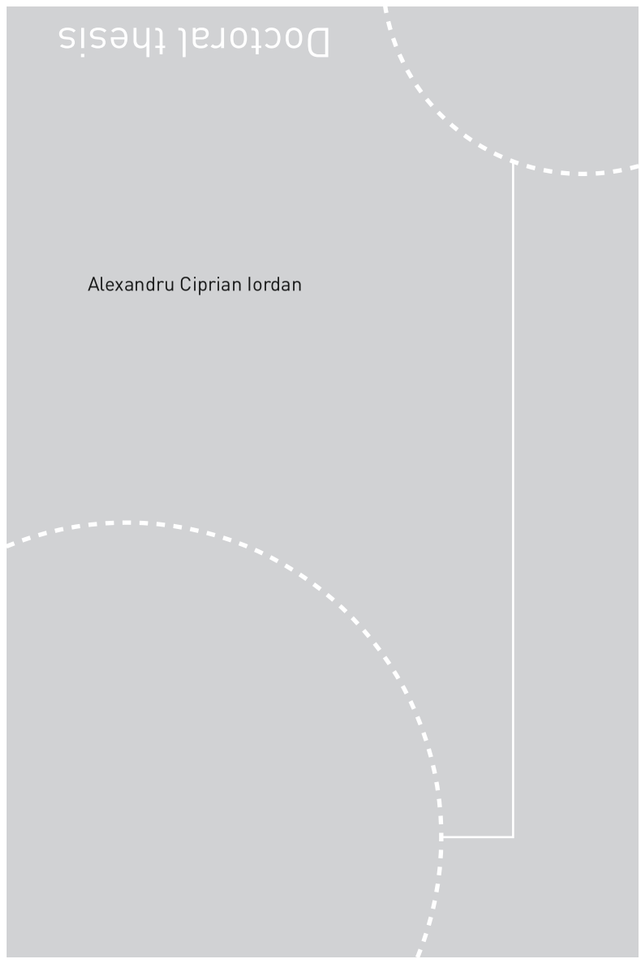
Doctoral theses at NTNU, 2017:175

Alexandru Ciprian Iordan

Improving the Energy-Efficiency of Task Based Programming on Chip Multiprocessors



ISBN 978-82-326-2422-5 (printed ver.)
ISBN 978-82-326-2423-2 (electronic ver.)
ISSN 1503-8181

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology
Thesis for the Degree of
Philosophiae Doctor
Faculty of Information Technology and Electrical
Engineering
Department of Computer Science

Doctoral theses at NTNU, 2017:175

NTNU
Norwegian University of
Science and Technology

Alexandru Ciprian Iordan

Improving the Energy-Efficiency of Task Based Programming on Chip Multiprocessors

Thesis for the Degree of Philosophiae Doctor

Trondheim, June 2017

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the Degree of Philosophiae Doctor

Faculty of Information Technology and Electrical Engineering
Department of Computer Science

ISBN 978-82-326-2422-5 (printed ver.)
ISBN 978-82-326-2423-2 (electronic ver.)
ISSN 1503-8181

Doctoral theses at NTNU, 2017:175

Printed by NTNU Grafisk senter

To my loving wife and our beautiful children.

Abstract

In the early 2000s, the superscalar CPU paradigm reached the point of diminishing returns mainly due to power requirements and overheating concerns. Faced with a constant demand for performance, hardware developers were in need of new ways to efficiently use the ever increasing transistor count predicted by Moore's law. The Chip MultiProcessors (CMPs) came as a natural solution to *the power wall*: several less complex and significantly less "power hungry" cores integrated on a single chip. In almost all ICT segments today, from High Performance Computing (HPC) to embedded devices, CMPs have become the architecture of choice. With this wide adoption of CMPs, software developers need to use parallel programming to fully exploit this architecture. Although parallelization can maximize the performance and energy efficiency of applications running on CMPs, it also comes with its own set of challenges. Among these, inherent management overheads that can account for sub-linear speedups and can increase the energy consumption of executions. Because of rising concerns for energy cost and battery life, much research and development today focuses on reducing power requirements and saving energy.

In this thesis, we investigate how parallel programming can be used to improve the energy efficiency of applications running on CMP systems. We focus on a programming paradigm called Task Based Programming (TBP). The base concept of the TBP model is that the programmer focuses on identifying and annotating pieces of code (tasks) which can be executed concurrently with other tasks. An important result of our work is an increased understanding of how computations, parallelization and energy consumption relate when executing on CMP systems.

Working in this direction, we use a simulation framework to allow for increased flexibility in design space exploration and noninvasive measurements. Unfortunately, the performance overhead of simulation is significant: simulating a parallel application can be 10000x slower than executing it on real hardware. In the first part of our research, we took it upon ourselves to try to solve this issue. We investigate the challenges of employing a sampling based technique to take advantage of the periodic behavior in TBP parallel applications. Our proposal is a simple 3-phase methodology that identifies only a small number of representative execution samples to simulate thus reducing the overall simulation time.

In the second part of our work, we look at parallelization as a mean to save energy on CMP platforms. We test and compare two TBP libraries, Wool and Intel TBB, focusing on the behavior of some basic TBP parallelization operations like task spawning, task synchronization and task stealing. We investigate the energy footprint of these parallelization overheads and the effect it has on the energy-efficiency of the executing system. We have identified that failed task steals amount for the largest overhead. To reduce their impact and improve our system's energy efficiency, we devised a new occupancy-aware policy for victim selection. This policy allows for a more informed decision when selecting the victim for task stealing and reduces the number of failed steals.

Preface

This doctoral thesis was submitted to the Norwegian University of Science and Technology (NTNU) in partial fulfillment of the requirements for the degree of philosophiae doctor (PhD). The work herein was performed at the Department of Computer and Information Science (IDI), NTNU, under the supervision of Professor Lasse Natvig.

This thesis consists of two parts. The first part consists of introduction, background, methodology, research process, a summary of papers and final conclusions. The second part is the main contribution, presented as a collection of six research papers.

Acknowledgements

No deed is achieved single-handedly and the work for this thesis is no exception. There are so many people that I need to thank for their support, guidance, or simply being there with me throughout my PhD experience.

First, I would like to thank my supervisor Prof. Lasse Natvig for his advice and guidance during my Norwegian adventure that became this thesis. From the first evening when I arrived in Trondheim and he picked me up from the airport bus, it was his help and constant support that made me stay on track. Thank you for being patient with my many ideas and (slow) investigations, but also for pulling me back when I was a bit too ambitious. I also extend my gratitude to my co-supervisors, Prof. Per Gunnar Kjeldsberg and Dr. Gaute Myklebust for providing valuable feedback and encouragement in my evaluation meetings.

I also owe a great deal of gratitude to Magnus Jahre for his help beyond and above my academic work. He was my "person to go to" for any issues that I faced, research or otherwise, and he always provided me with good insight. Thank you for the countless *slabberas*[1] meetings and for helping me go over the cultural barier.

I would like to thank my colleagues at the department of Computer and Information Science throughout the years: Kostas, Dragana, Asbjorn, Stefano, Ian, Nico, Yaman, Jan-Christian, Abdullah, Nikita, Juan Manuel, Odd Rune and many others. Thank you for providing a great working environment with countless interesting, engaging and diverse discussions.

I would like to thank Ana Lucia Varbanescu for first suggesting me the PhD position at NTNU. Her support and guidance in preparing for the interview was invaluable as well as all the advices she provided ever since.

I also need to thank Prof. Radu Varbanescu for introducing me to all the wonders and perils of academic research. It was his support during my bachelor and master studies in Romania that got me to the point where I wanted and was able to do PhD work.

Finally, I would like to thank my family. To my parents and parents-in-law thank you for all the help and support in making sure I do not feel alone far away from my home country. Your visits and care packages over the years have meant a lot for me. However, to my loving wife Cristina I owe maybe the most. Without her relentless support before and during my PhD work, none of it would have been possible. And also I should not forget to mention our two beautiful children, David and Matei, who made sure I had something to do outside my work.

[1] Norwegian: slabberas - pleasant conversation over tea or coffee

Contents

List of Tables

List of Figures

Acronyms

ALU Arithmetic-Logic Unit.

ANTT Average Normalized Turnaround Time.

API Application Programming Interface.

BPS Bulk Synchronous Parallel model.

CCI Cache Coherent Interconnect.

CMP Chip Multi Processor.

CPI Cycles Per Instruction.

CPU Central Processing Unit.

CUDA Compute Unified Device Architecture.

DDR Double Data Rate memory.

EDaP Energy Delay-Area-Product.

EDP Energy Delay-Product.

FinFET Fin Field Effect Transistor.

FPU Floating Point Unit.

GPGPU General Purpose computing on Graphical Processing Units.

GPU Graphical Processing Unit.

GTS Global Task Scheduling.

HMP Heterogeneous Multi-Processing.

HPC High Performance Computing.

HSAIL Heterogeneous System Architecture Intermediate Language.

ICT Information Communication Technology.

IDI Institutt for datateknikk og informasjonsvitenskap (Department of Computer and Information Science).

ILP Instruction Level Parallelism.

IME Fakultet for Informasjonsteknologi, Matematikk og Elektroteknikk (Faculty of Information Technology, Mathematics and Electrical Engineering).

IPC Instructions Per Cycles.

ISA Instruction Set Architecture.

LLC Last Level Cache.

LLVM Low Level Virtual Machine.

NTNU Norges Teknisk-Naturvitenskapelige Universitet (Norwegian University of Science and Technology).

OpenCL Open Computing Language.

OpenGL Open Graphics Library.

OS Operating System.

PCI Peripheral Component Interconnect.

QPI Quick Path Interconnect.

SIMD Single Instruction, Multiple Data.

SMT Simultaneous Multi-Threading.

SoC System on Chip.

STP System Throughput.

TBB Threading Building Blocks.

TBP Task Based Programming.

VLIW Very Long Instruction Word.

Chapter 1

Introduction

1.1 From single-core to multi-cores

Ever since the beginning of computing, the responsibility for increasing application performance has shifted between hardware and software designers. With the technological advances of the 1980s, many hardware constraints (size and speed of memories, speed and complexity of processing units etc.) from previous decades were reduced and application development became easier. By applying Dennard's scaling rules [64], transistor dimensions shrank by 30% each CPU generation which in turn allowed for an impressive 55% increase in performance every 18-24 months [91]. With more transistors that could be cramped on single chip, faster and more complex CPUs were produced. In this period application speed-up came almost for free, as every new generation of processors increased clock speed allowing for more operations to be executed per unit of time.

Early into the 21st century it became apparent that even if Moore's law was still holding true, Dennard's scaling rules cannot continue to boost performance. Facing issues due to high power requirements, high memory latencies and high "costs" of extracting Instruction Level Parallelism (ILP), CPU designers needed a new approach. Around 2002 they moved towards using the high transistor count to build multiple, more energy-efficient, cores on a single die [79, 144]. Evolving through several generations, CMPs have become the many-cores, multithreaded, shared-cache CMPs of today[143, 162]. Early generations of CMPs were using two conventional superscalar cores stitched together to use a common memory bus. Later designs used more tight integration and were able to fit more than two processing units (PU) on a chip which were sharing an L2 cache. This allowed for a big speedup in intra-core communication, but it also introduced new type of challenges like cache partitioning, fairness and Quality of Service [68, 69, 104]. The latest generation includes simpler and more energy-efficient cores capable of simultaneously multithreading (SMT). Figure 1.1 summarizes this CMP evolution.

However, the idea of using multiple processing units to improve performance was not new. Mostly used in advanced research and HPC, multi-processor systems have been

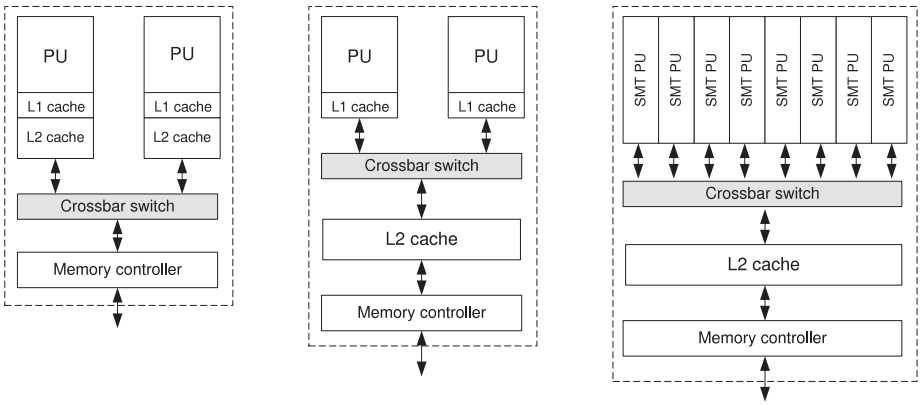


Figure 1.1: Evolution of CMPs from early generation (left) to recent design (right).

around for decades. The advantages and disadvantages of using them for general purpose computing has been discussed since the late '60s, when Gene Amdahl made his case for the use of the single processor [27]. He put forward a formula, which became known as *Amdahl's law*, to calculate the theoretical maximum speedup of an execution when using multiple processors. By Amdahl's law only *embarrassingly parallel* applications can achieve the maximum speedup when running on multiple cores. However, modern CMPs employ techniques like task switching to increases CPU throughput and make better use of the hardware.

There are four main technological and economical constraints that have pushed the shift to CMPs: the power wall, the memory wall, the ILP wall and the complexity wall. The next sections briefly describe each of these.

1.1.1 The Power Wall

Dynamic power (the power required by an active transistor) is defined as:

$$power_{dynamic} \sim \ AF \cdot \ C \cdot V_{dd}^2 \cdot f$$

where *AF* is the activity factor, *C* is the total load capacitance, V_{dd} is the supply voltage and f is the clock frequency [107]. When scaling down the size of a transistor, its supply voltage can also be reduced while its clock speed can be increased. Looking at the formula above, this means that we can mitigate the impact on power requirement of a higher clock speed by reducing the supply voltage. Using this technique, chip manufacturers have succeeded in producing ever more complex and faster single-core CPUs through the 90s (see Figure 1.2). Power requirement was manageable, but expensive packaging and powerful cooling solutions were required to keep these chips working. The benefits in

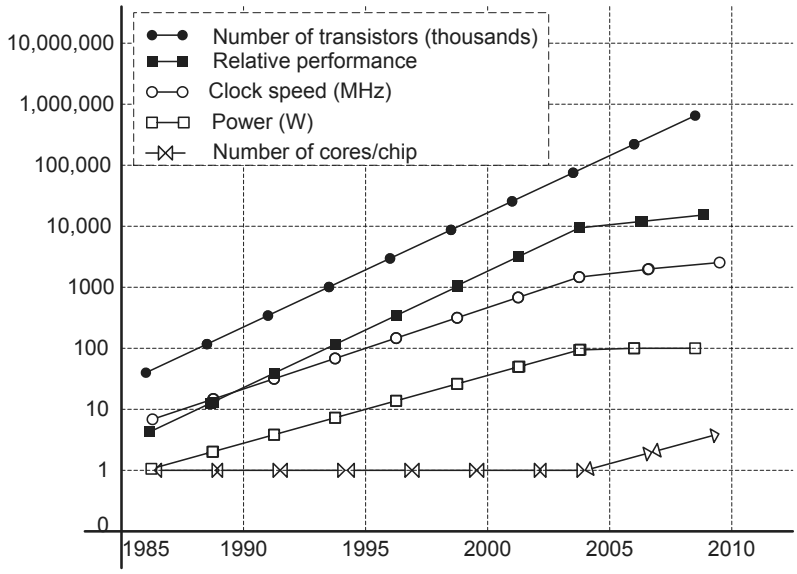


Figure 1.2: Technological trends for microprocessors. Simplified version of Figure 1 in [78].

terms of power from technology scaling faded away when going from the 90 nm feature size to the 65 nm one (around the year 2004) for several reasons. First, in smaller feature sizes leakage currents increase. Second, the need to extract more ILP to keep the deep pipelines full was increasing the complexity of the circuits and the activity factors of the transistors. Third, in the aggressive competition to release the fastest CPU chip, area and frequency were scaled over what was recommended by Dennard's scaling rule. All in all power dissipation went beyond practical limits and it was said that single-core design has hit the *power wall* [88].

As a natural solution, CMPs pack several less complex and significantly less "power hungry" cores on a single chip. The individual performance of a core is well below that of a single-core CPU, but the aggregate performance of the chip can be better if the software is designed to run in parallel. While CMPs allow designers to continue boosting performance, they are expected to run eventually into the same power wall. A study by Esmaeilzadeh et al. [71] estimates that when reaching the 8nm feature size approximately 50% of a fixed-size chip will have to be powered down in order to mitigate its power requirements.

Figure 1.2 shows the evolution of CPUs and the shift towards CMPs. While Moore's law still holds true (the number of transistors is increasing exponentially), the CPU performance, clock speed and power requirement have a different trend since 2004.

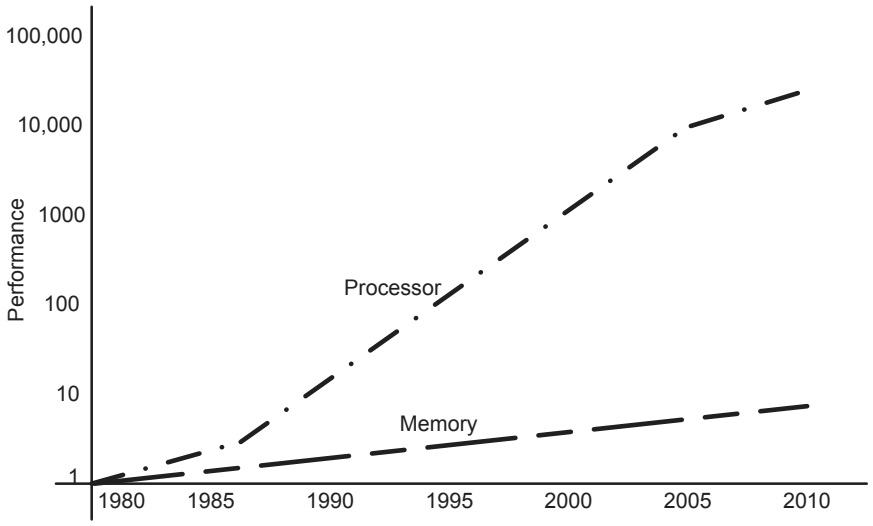


Figure 1.3: CPU performance versus memory latency [91].

1.1.2 The Memory Wall

For about 20 years, processor performance has increased almost 55% per year while memory latency decreased only 6-7% per year [91, 147] (see Figure 1.3). What this means is that every new generation of CPU could process the data much faster than it can be retrieved from main memory. This growing performance issue prevents users from utilizing the CPU at its fullest and is referred to as the *memory gap*. Wulf and McKee discussed this memory gap in their 1995 paper and predicted that there will be a point where CPU development will face a *memory wall* [181]. The main solution for this problem was the introduction of caches hierarchies. As the gap widened, deeper hierarchies were needed to keep up with the memory access requests of the CPU. In 2002, with a three level cache hierarchy and very aggressive mechanisms for data prefetching, it was said that single-core design has hit the *memory wall*.

The memory wall also affects CMPs, but since each core runs at a lower frequency the gap slowed down a bit. Also, for CMPs the main issue is to hide the bandwidth demand rather than the latency. Doubling the number of cores and the amount of cache every CMP generation results in a corresponding doubling of off-chip memory traffic. This implies that the rate at which memory requests must be serviced also needs to double [160].

1.1.3 The ILP Wall

With single-core CPUs becoming faster and faster trough out the 90s, their pipelines deepened and ever more ILP was needed to keep them full. Studies showed that high parallelism can only be extracted at very high hardware costs [177]. In the beginning of the 21st century common techniques for exploiting ILP became insufficient and it was said that single-core design has hit the *ILP wall*. CMP design alleviate this issue since each core has a shallower pipeline and requires fewer independent instructions to keep them busy. Also, in a multi-core design the programmer can provide higher level of parallelism (task level, thread level).

1.1.4 The Complexity Wall

The very complex single-core CPUs from the late 90s required a significant effort to design, develop and test. Only a handful of companies could afford the man power and infrastructure required for such a task. This was referred to as the *complexity wall*. Since so much effort was channeled towards boosting performance of the single-core CPUs, innovation was stalling.

In contrast, CMP design allows the reuse of a core design within a chip. Also, new design challenges like cache partitioning and fairness open possibilities for innovation and differentiation among developers.

1.2 The parallelization issue

In terms of software development, the shift from single-core to CMPs caused a great deal of pain [169]. Most existing applications as well as all the new ones emerging from every computational field had to be parallelized and optimized for running on multi-core processors. This is not a simple task: selecting the right architecture, choosing a parallelization technique and tuning the code for the specifics of the hardware platform (employing data distribution strategies etc.) are all very difficult [173]. But failing to do so would make CMPs highly inefficient.

Even though there is a large collection of parallel programming models, languages and compilers the software developers are still struggling with the parallelism requirements posed by the multi-cores. With CMPs making their way in every electronic device on the market today, this ”programmability gap” has become a concern in both hardware and software communities. Both academia and industry have attempted to bridge this gap and provide developers with parallel libraries that aim at improving application portability and programming efficiency [52, 74, 83, 119, 121, 150]. Even if in the end the parallelization succeeds, the added management overheads can account for sub-linear speedups and can

increase the energy footprint of the application. Understanding and limiting these overheads is a necessary step towards scalable and more efficient runtime parallel libraries.

1.3 Research Questions

Investigating ways how use or improve parallel programming on CMPs to increase a system's energy efficiency are our main research goals. Our approach relies on the following three research questions:

RQ1 What is the potential for energy saving for the TBP model on a multi-core system?

RQ2 How do basic operations in TBP parallel code (task spawning, task migration, synchronization etc.) affect the energy footprint of the execution? What are the performance/energy trade-offs?

RQ3 How can we improve the energy efficiency of TPB programs?

ExtraRQ How can we reduce long simulation times of deterministic architectural simulators like M5/GEM5?

1.4 Thesis outline

The remainder of this thesis is organized in the following way:

- Chapter 2: - theoretical background related to the contributions in this thesis. The chapter gives an overview of the main theoretical concepts that our research is based on.
- Chapter 3: - the methodology used to produce the research results.
- Chapter 4: -the research progress. This chapter presents how we produced all our papers during the doctoral research. The papers that are not included in this thesis are also discussed.
- Chapter 5: - research results summary. This chapter overviews the papers included in the thesis and highlights the contribution of each author.
- Chapter 6: - contribution summary and concluding remarks. This chapter reviews the thesis contributions in relation to the research questions. It also describes the challenges and potential directions of future work.
- Appendix A: - enclosed selected papers.

Chapter 2

Background

This chapter provides some of the fundamental concepts required to understand the research in the papers that are part of this thesis. In section 2.1 we introduce the necessary background on performance and energy-efficiency measurements. Section 2.2 presents the evolution of CMPs from early generations to present designs. The *many thin cores or few fat cores* debate is outlined in section 2.3 together with some representative architectures for each class. And in section 2.4 we discuss the parallelization model and the implementations that we focused on in this research.

2.1 Metrics

When evaluating a new design or a novel research idea, the use of correct metrics is paramount. Poorly chosen metrics can lead to incorrect conclusions and can steer a researcher in the wrong direction [70]. This sections gives a very brief overview of some performance and energy-efficiency metrics, most of which are used in papers included in this thesis.

Energy, measured in Joules, is the most fundamental metric in power and energy studies, particularly those of mobile platforms where it correlates with battery life. Energy consumption is also of great importance for facilities like data centers, where it is among the top operating costs.

Power, measured in Watts, is the rate of energy dissipation. Power and its related metrics, like areal power density, are meaningful in studies of current delivery, voltage regulation and thermal dissipation.

Energy-delay product (EDP) was first proposed by Horowitz et al. [95] as a metric to evaluate differences between energy savings techniques for digital circuit designs. With low energy and fast runtimes in mind, EDP will give a better ranking to a faster execution of the same instruction mix using the same amount of energy or to the same execution

time but using less energy. EDP offers equal weight to both performance and energy efficiency, but alternative metrics have been suggested that give a higher priority to the performance component. These are energy-delay-squared product (ED^2P) and energy-delay-cubed product (ED^3P). ED^2P and ED^3P are used mostly in the high-performance field, where performance gain are more important than energy efficiency. The energy-delay-area product (EDAP) and energy-delay-area-squared product (EDA^2P) add the cost of the component to the metric. Die cost is roughly proportional to the square of the area [91], so EDA^2P is a good metric to use while designing a CMP. At the system level, EDAP may be preferred since fixed system costs such as memory and I/O reduce the overall dependency on CMP cost.

Execution time is argued by most as the only reliable measure of performance [91]. Hennessy and Patterson define *wall clock time* as the time required to complete an application, including operating system tasks, disk accesses and other I/O operations [91]. In a system running several programs simultaneous, the wall clock time of one program may depend on the progressing of another. That is why *CPU time* is defined to include only the time a CPU uses to execute the instructions of one program.

Cycles Per Instruction (CPI) and its reciprocal *Instructions Per Cycle* (IPC) were the most used metrics for CPU performance in the single-core era. CPI measures the average number of cycles it takes to execute an instruction for an application or application section.

$$CPI = \frac{\sum_i (IC_i \cdot CC_i)}{IC} \tag{2.1}$$

In Formula 2.1, IC_i is the number of instructions for a given instruction type, CC_i is the average number of clock-cycles needed to execute that instruction type and IC is the total instruction count. The summation is done over all instruction types for a given execution. IPC is calculated as 1/CPI.

CPI and IPC can be very misleading metrics when applications execute in parallel for many reasons [70]. For example, access control mechanisms like spinlocks will increase the instruction count without actually making any forward progress in the execution. This means that the instruction count will be very high which will lead to an erroneous performance result. To better quantify the performance of such applications, work oriented metrics are needed. One example for databases is *Transactions Per Second* which measures the number of useful database transactions per second.

Multi-core CPUs also allow for multiprogram workloads to be executed and such situations require a new type of metrics. Eyerman and Eeckhout propose two metrics for this purpose: a user-oriented metric called *Average Normalized Turnaround Time* (ANTT) and a system-oriented metric called *System Throughput* (STP) [72]. ANTT is defined as the arithmetic average across the programs' normalized turnaround times (the user-perceived slowdown during multiprogram execution relative to the single-program execution). STP is defined as the sum of the normalized progress rates (the programs' progress during multiprogram execution relative to the single-program execution) across all jobs.

When evaluating a new technique, most often the *speedup* achieved when compared to a baseline is of interest. This is calculated according to Formula 2.2.

$$Speedup = \frac{Performance_{new}}{Performance_{baseline}} \tag{2.2}$$

2.2 Chip Multiprocessors

As Gordon Moore described in 1965, the decrease in manufacturing costs for dense integrated circuits allowed the number of transistors per chip to double every year [138]. He projected this trend will continue for at least 10 more years. In 1975 he revised this forecast to doubling every second year. With this exponential increase in transistor count, CPUs have seen tremendous improvements for several decades (see Figure 1.2). In early 2000s, Dennard's scaling rules could no longer provide the necessary reduction of a chip's power requirements and a shift towards multi-cores occurred in CPU design.

To facilitate our latter discussion, this section will give a brief description of the evolution of CMPs from early generations to more current homogeneous and heterogeneous designs. It will also present some representative examples targeting mostly similarities and differences on an architectural level.

2.2.1 Evolution of CMPs

The constraints described in Section 1.1.1 - 1.1.4 did not take the tech world by surprise. All of them were foreseen [88, 177, 181] and various architectural solutions were put forward, many of which described a CMP design. As early as 1996 Olukotun et al. made their case for the development of single-chip multiprocessors [144]. Several research teams proposed architectural designs for CMPs like the Hydra [90], the Pirahna [33] or the MAJC [168] years before the industry made the switch. The need for the design change was understood by the main CPU developers and IBM [167], Sun [106], Futjitsu [133] and Intel [48] announced their intention of developing CMP chips.

For many first generation CMPs (like the dual-core UltraSPARC-II [106] or the dual-core POWER-4 [167]), the cores were "carbon copies" of single-core designs. Reusing core designs was a natural incremental step for such a design change, however it was in no way a scalable solution. The off-chip data paths were the only shared resource among the two cores (see Figure 1.1). These CMPs were primarily designed for the HPC market and their primary goal was a reduction in volume: two to four cores can fit where only one could resulting in a higher performance per unit volume. Power savings were also achieved since a system using a CMP would need only one power source and would not require the high-speed interconnection several single-core systems would [143].

For the second generation CMPs, CPU developers tried to improve performance by sharing a L2 cache among cores. With a L2 cache shared among several execution units, designers needed to ensure cache coherency. This approach allowed executions with large input sets to be splitted up and performed in parallel on a single CPU rather than on several interconnected systems resulting in a performance boost. Core designs were still based on stripped down versions of older single-core implementations.

If you were to double the number of cores each generation, the power requirement of each core needed to be halved in order to maintain the same power envelope. With Dennard's scaling rules no longer providing the desired reduction in power for feature sizes beyond 65 nm [88], techniques like clock gating and frequency scaling were employed. In addition, off-chip bandwidth per core was also a design issue. As the packaging costs per pin increase significantly with pin count, the focus has been to increase the per-pin bandwidth [162]. These issues could only be addressed if the cores were designed specifically for CMPs. Sun's Niagara processor was among the first third generation CMPs (see Figure 1.1) [110]. Niagara was a 32-way CMP that used 8 cores each with a private L1 cache and a shared L2 cache. Niagara's developers used simple single issue cores with a short pipeline. With this low complexity level of the cores, the per-core as well as the overall power requirement was significantly reduced when compared to a superscalar CPU. Such design could only be fully exploited by applications with a high level of thread level parallelism.

2.3 Homogeneous or heterogeneous?

One of the main challenges CMP designers face today, is how many and how complex the cores should be on a chip. The industry has failed to find a "silver bullet" when it comes to the *many thin cores or few fat cores* debate. Amdahl's law describes how the serial part of an application can have a big impact on the speedup that an application can achieve through parallelization. To perform well for both serial and parallel executions, designers need to hit a balance between the complexity of cores and the number of cores that can fit on a chip [93]. The development simplicity and ease of programming has made symmetric homogeneous CMPs the design of choice for general purpose CPUs. However, studies have shown that a heterogeneous approach could bring a substantial improvement in energy efficiency, with minimal loss to performance [112]. Also, heterogeneous chips can accommodate both large and complex cores best fitted for single-threaded, high ILP executions as well as simple and energy efficient cores targeted for high throughput applications [113]. Even more performance improvements have been forecasted if heterogeneity is pursued further and custom cores are developed specifically for CMP design [114].

2.3.1 Homogeneous architectures

In a *homogeneous* CMP all cores are of the same type, both in instruction set architecture (ISA) and performance. This means that any thread can be executed on any of the cores with the exact same results. From a chip developer's point of view, such designs are "simple" since the same core design is replicated several times on the die. From a software developer's point of view, programming this type of CMPs is more straightforward since there is only one instruction set to think about, only one type of compute resources and it makes load balancing relatively easy.

In the next subsections we will present two modern CMP architectures in this class. We will emphasize the architectural features that brought improvements, both in performance or energy-efficiency when compared to previous CMP generations. We will also highlight the main differences that make these architectures stand out.

Intel Haswell

Haswell is the codename for Intel's forth architectural generation of Core CPUs. It was first introduced in 2011 at Intel Developer Forum [21] and the first chip that used this architecture was released in 2013. The list of CPUs that Intel has released using this architecture is very long and it spans from desktop PCs, HPC servers and for the first time mobile ultrabooks.

Haswell chips support from 2 to 18 cores which coupled with Intel's Hyper-threading means that they can handle up to 36 threads. Each core integrates 4 ALUs, 2 vector integer ALUs, 2 FPUs, 3 address generation units, 2 branch execution units, all integrated in a 14- to 19-stage instruction pipeline. The highest-level shared cache is usually referred to as the *Last Level Cache* (LLC). For most Haswell chips L3 is the LLC, but there are variants like the R-series desktop processors which include a 128 MB shared L4 cache level. Even though the LLC is not a unified cache, Intel designed it so that it will scale linearly with the number of cores (one cache bank per core). Most Haswell chips also include a GPU from Intel's HD Graphics family, but this will not be the focus of this section.

First introduced by Intel in 2008 to compete with AMD's HyperTransport, the Quick Path Interconnect (QPI) is a point-to-point CPU interconnect. It is designed to link processors in what Intel calls a Quick Path Architecture. Separate QPI link pairs will connect several CPUs and several IO hubs in a network. The cores, the LLC, the memory controller, the QPI interconnect and the PCI-express buses are all linked by a ring bus (see Figure 2.1).

Built on the same 22 nm process as previous generation, Haswell's improvements in power and performance are not due to miniaturization. To improve on its predecessor's power requirements, Intel has implemented a new power management framework which allows the CPU to handle device driver events in scheduled batches rather than real time.

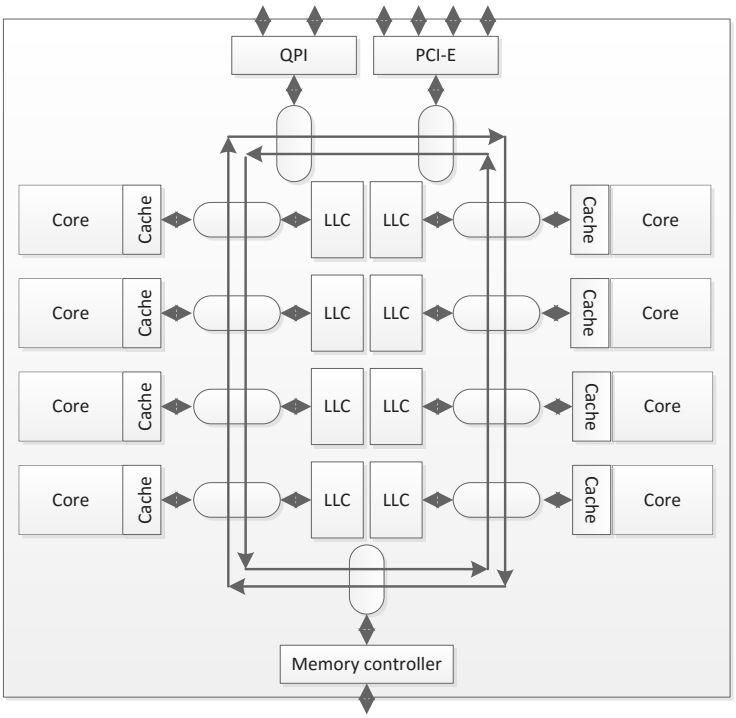


Figure 2.1: Intel Haswell architecture, simplified block diagram.

In this way, the CPU can spend more time powered down between bursts of activity. To boost Haswell's off-chip bandwidth, Intel has introduced support for DDR4 memory.

For programming its CMPs, Intel provides software developers a set of highly optimized libraries like Intel Integrated Performance Primitives (IPP) [22] or Intel Math Kernel Library (MKL) [23]. Intel Cilk Plus [20] is a general-purpose programming language, originally developed at MIT and later acquired by Intel. It extends the C and C++ programming languages with constructs to express parallel loops and fork-join mechanisms. Intel Threading Building Blocks (TBB) [24] is a template library that supports scalable parallel programming using standard C++ code. In addition to these proprietary solutions, developers can still use low-level multi-threading primitives or high-level generic solutions like OpenMP [52] or MPI [83].

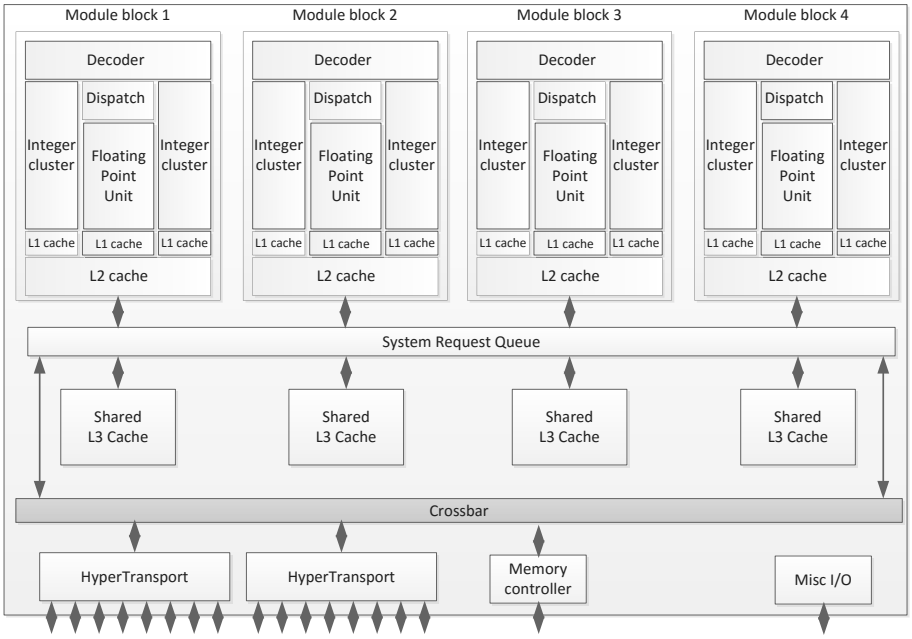


Figure 2.2: AMD Piledriver architecture, simplified block diagram.

AMD Piledriver

Piledriver is the codename for AMD's second generation architecture in the Bulldozer family. AMD refers to the chips in the Bulldozer family as *Accelerated Processing Units* because they enclose both the CPU and GPU in the same package. The first generation Bulldozer chips used a new design from the ground up, rather than being an incremental development of an earlier processor. Improvements in branch prediction and scheduling for both floating point and integer operations give Piledriver a performance boost over its predecessors. Its energy efficiency has been increased by switching from soft- to hard-edge flip-flops. Flip-flops are used to store data or a state and the information is latched at the rising edge of the clock. This means that they are very sensible to clock jitter. Soft-edge flip-flops are more tolerant to this jitter and are able to function even if the clock signal spills over the edge. On the other hand, hard-edge flip-flops are smaller and less complex than their soft counterparts, but using them involves an extra effort to account for the timing margin.

Piledriver uses the same *module* design introduced by the first generation Bulldozer CMPs. Each module consists of 2 integer clusters, 1 floating point cluster capable of SMT, dedicated L1 data cache for each cluster and shared L2 cache (see Figure 2.2). Each integer

cluster contains an integer scheduler in addition to the integer execution units (2 ALUs and 2 address generation units), while the floating point cluster contains 2 symmetrical 128-bit fused multiply-add pipelines which share a scheduler. Because of its dedicated hardware in the integer modules, this design is more effective while executing two threads consisting of integer operations when comparing it to an SMT core. The disadvantage of this architectural approach is that a single thread execution consisting of integer operations can only use one integer cluster, resulting in underutilization of the resources. Both integer and floating point clusters in a module share the L1 instruction cache and the decode-dispatch logic. All modules share a non-unified L3 cache connected via the System Request Queue as well as an integrated memory controller connected via a crossbar interconnect (see Figure 2.2). Like Intel's QPI, AMD also provides a point-to-point link called HyperTransport to interconnect several processors on a motherboard.

Like Intel, AMD provides a set of libraries called AMD Compute Libraries to help developers in programming its CMPs [2]. This set includes both GPU (dBLAS, dSparse, dFFT) as well as CPU libraries (BLIS, libflame). Developers can also use low-level multithreading primitives or high-level generic solutions like OpenMP [52] or MPI [83].

2.3.2 Heterogeneous architectures

Heterogeneous or asymmetric systems are not a new concept. For the room-size computers of the '60s and '70s, a cost-effective way to increase compute power was to add a second processing unit to the system. The single-CPU operating systems of the time were initially extended with minimal support for the second CPU so it could only execute user programs. This means the CPUs were not able to perform the same tasks which is why those systems were ranked as asymmetric. Operating such systems required an extensive knowledge and were only used in research institutes and large universities [3].

A CMP is called heterogeneous if it incorporates dissimilar cores, usually featuring specialized processing capabilities that can handle particular tasks. The heterogeneity or asymmetry of a CMP can be pursued both in hardware and software. In hardware, cores of different size and features can be placed together on a single die. These architectures provide the flexibility of scheduling the workload to those processing units that best match the system's requirements for performance or energy-efficiency [6, 139]. Modern mobile chips are implementing such designs as single-ISA heterogeneous CMPs [4]. In software, the ISA, OS and even programming model of the various cores can be different [124, 125]. Pursuing heterogeneity in both software and hardware at the same time with heterogeneous-ISA CMPs promises to give out even more benefits [174].

The design of heterogeneous chips brings up a completely different set of challenges that span across the system stack [56, 87, 126]. However, heterogeneous CMPs hold great potential in optimizing both sequential and parallel computing [137]. To achieved it, chip designers need to work towards tightly integrating hardware multithreading, many cores, SIMD units, accelerators and on-chip communication systems [46]. At the same time,

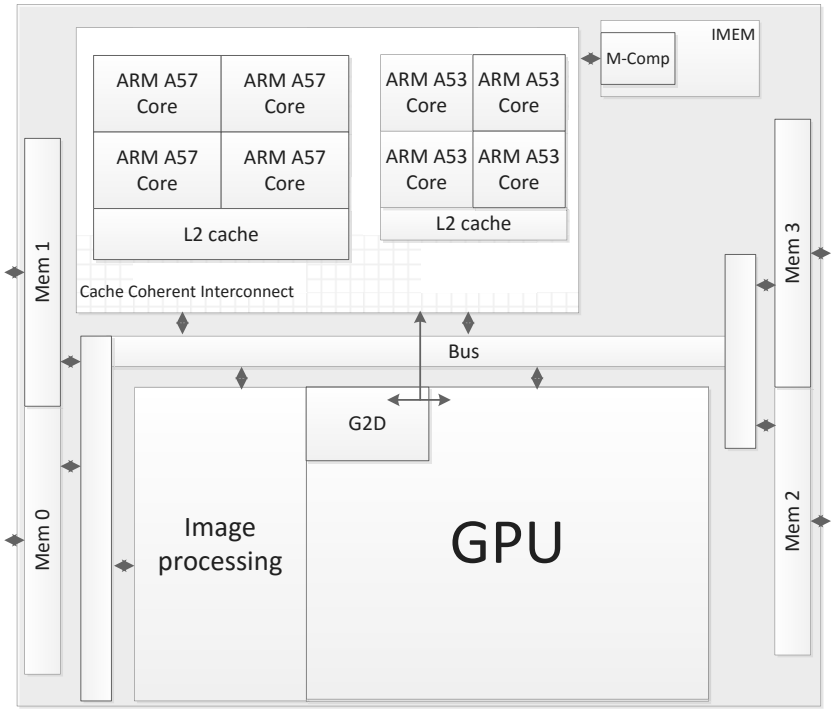


Figure 2.3: Exynos 7 architecture, simplified block diagram.

programmers will need to exploit parallelism, orchestrate computations and manage data locality at several levels in order to achieve reasonable performance [47].

In the next subsections we will try to give a brief overview of some recent or under-development heterogeneous chip designs. We focused of those architectures that seemed to be the most successful in today's market, like ARM big.LITTLE CPUs. We discuss advantages and challenges of systems using only heterogeneous CPUs but also CPU plus accelerators. And finally we also present research to develop the next generation of heterogeneous CMPs for the HPC sector.

Heterogeneous CPU

Samsung's Exynos family, like other chips designed for mobile platforms, are part of a class called System on Chip (SoC). This mean they incorporate functional units and logic capable to perform several tasks like: compute jobs (CPU), graphics (GPU), image processing, audio processing, radio modems etc. In this Exynos overview we will only focus on the CPU block and its big.LITTLE heterogeneous architecture. Samsung's implemen-

tation of the big.LITTLE design is known as *heterogeneous multi-processing* (HMP) and it relies on a scheduling mechanism developed by Linaro (a consortium of ARM vendors) called Global Task Scheduling (GTS) [6]. Figure 2.3 is an abstract simplification of the Exynos 7420 SoC, showing only some of the largest blocks present on the chip.

The CPU block is comprised of 8 cores arranged in two clusters. The big cluster includes 4 ARM A57 cores clocked at 2.1 GHz while the LITTLE one consists of 4 ARM A53 cores clocked at 1.5 GHz. The clock frequency is not the only difference between these two core architectures. ARM A57 is a 3-wide/3-issue CPU with a 15+ stage pipeline while the A53 is a 2-issue in-order architecture with an 8 stage pipeline. Both support 64-bit operations and the same ARM ISA. The two clusters are connected via an ARM Cache Coherent Interconnect (CCI) which is the key component in the heterogeneous setup of ARM's big.LITTLE design. The CCI enables cache coherency among the two core clusters and allows a thread's working set to migrate from one cluster to the other [4]. Using GTS, the OS manages the workloads on the clusters and the task migration between them [1, 5]. When used in mobile devices, the scheduling is optimized to use the LITTLE cores as much as possible and use the big cores only for "heavy lifting" [6]. By matching a thread's computational requirements to the core that best fits its needs, this heterogeneous architecture allows its resources to be better provisioned. In this way it achieves a higher energy efficiency for a wider range of applications.

In addition to the CPU clusters, CCI has 3 more ports to allow the creation of a group of cache coherent devices. First such port is shared by 2 graphics blocks: a 2D graphics accelerator called Fully Integrated Mobile Graphics 2D (G2D) and the GPU. Having the GPU sharing the CCI port is a departure from ARM's guidelines, where we usually see the GPU using 2 ports of the CCI. However, Samsung also chose to not connect the GPU to the memory controllers via the CCI, but rather through the memory interface bus. This has relieved the bandwidth demand on the CCI and has allowed Samsung to clock the CCI to a lower frequency (up to 532 MHz) rather than half the DRAM frequency as ARM recommends [4]. The next CCI port is taken by a new type of logic block in the mobile space: a memory compressor. Called the Exynos Memory Compressor or M-Comp, this hardware was designed by Samsung to support and optimize the DRAM memory compression mechanism included by the Android 4.4 kernel. The memory compressor is part of a larger block called IMEM that contains other elements such as hardware cryptographic accelerators [4]. The last CCI port is reserved for ARM's debugging and trace system of SoCs.

With the Exynos 7420, Samsung is pursuing energy efficiency more than performance. By using a 14 nm FinFET process and by not pushing the clocks too high, Samsung managed to obtain a 35%-45% reduction in power requirement when compared to the Exynos 5433 predecessor [4]. This means that the 7420 SoC can be used in very thin mobile devices that don't have the ability to dissipate much heat or the room to accommodate a large battery. ARM's big.LITTLE heterogeneous design gives Exynos 7420 the ability to better match computational requirements to hardware resources improving its energy efficiency.

Heterogeneous CPU with GPU acceleration

In the last decade, mainstream computer systems usually included not only CPUs as processing elements. The most prevalent such processor has been the GPU, originally designed to perform specialized graphics computations in parallel. Over time, GPUs have become more powerful and more generalized, allowing them to be applied to general purpose parallel computing tasks (known as GPGPU) with excellent power efficiency [28, 85, 129].

Because CPUs and GPUs have been designed as separate processing elements, they are not optimized to work together efficiently on compute workloads. In the early days of GPGPU (around 2005), computational problems needed to be expressed as graphics primitives supported by one of the two major graphics APIs, Open Graphics Library (OpenGL) or DirectX. The need for this translations was removed by the advent of general purpose programming languages such as the Compute Unified Device Architecture (CUDA) and the Open Compute Language (OpenCL) [28, 53, 65].

Difficulty in programming these platforms is not the only challenge that needs addressing. A program running on the CPU queues work for the GPU using system calls through a device driver stack managed by a completely separate scheduler. This introduces significant dispatch latency, with overheads that make the process worthwhile only when the application requires a very large amount of parallel computation. To fully exploit the capabilities of these parallel execution units, it is essential for hardware designers to re-architect computer systems and to tightly integrate them [28].

In order to enable GPGPU in its designs, ARM supports the OpenCL language for both its CPUs and GPUs. The OpenCL framework enables heterogeneous execution across different types of processing units like CPUs, GPUs, digital signal processors (DSPs)or other accelerators [17]. In this model, a host CPU is running the code that sets up the *compute devices* (GPUs, DSPs or other CPUs) and schedules the *kernels* (computational tasks) for execution. The host is also responsible for managing the data transfers to and from the compute devices.

Prakash et al. [154] explore the energy-efficiency benefits of Samsung's Exynos 5422 SoC and discuss the trade-offs of CPU heterogeneity. The 5422 SoC implements the same HMP run-state migration as the Exynos 7420, using four ARM Cortex A15 cores in the big cluster, four ARM Cortex A7 cores in the LITTLE cluster and a Mali T628 GPU [13]. The GPUs in ARM's Mali T600 Midgard series were the first in the embedded world to fully support the complete OpenCL profile, which means that the 5422 can extend and accelerate processing with the GPU. The Mali T628 has six shader cores, four of which can be used for general purpose computing. Each shader core contains a multi-threaded processing engine called a *tri-pipe* made up of a load-store pipeline, two arithmetic pipelines and a texture pipeline (used only for graphics jobs). The arithmetic pipelines have a mix of scalar and vector ALUs that execute a single long instruction word (VLIW design). The results in [154] show an average of 19% increase in performance by using both CPU and GPU as oposed to using only CPU or only GPU. At the same time, judicious

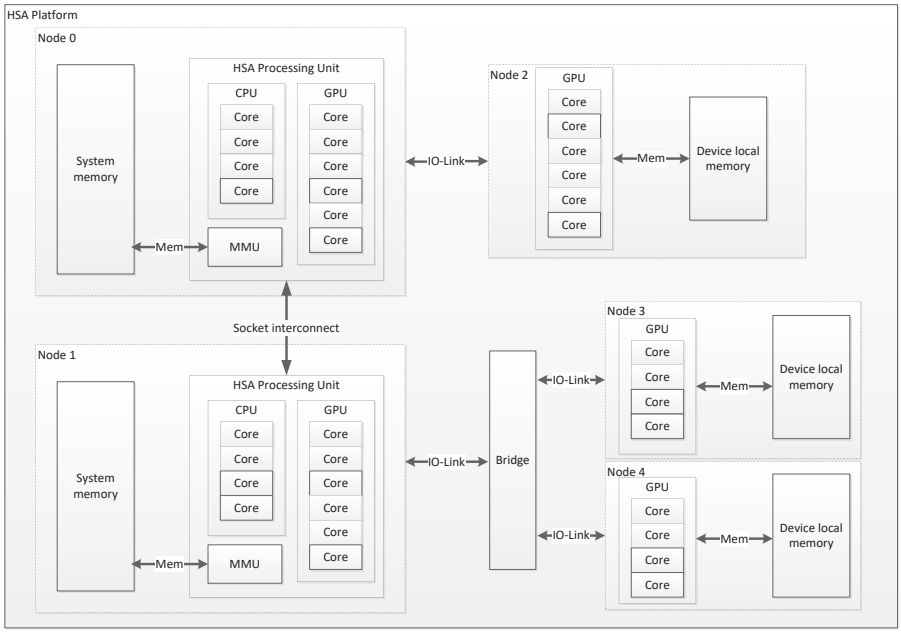


Figure 2.4: Example of a HSA multi-socket CPU-GPU design, simplified block diagram.

selection of operating frequencies for both CPU and GPU can lead to an average of 36% improvement in energy efficiency. However, we need to note that all these results have been achieved through statically partitioning the workloads across CPU and GPU cores.

Delporte et al. [63] have proposed an optimization scheme called *heterogeneous platform accelerator* (HPA) that will allow generic applications to dynamically exploit all available computational units. Their approach is language agnostic and relies on the Low Level Virtual Machine (LLVM)'s Just-In-Time (JIT) compiler [118]. As the executions starts, a performance monitor is constantly analyzing the executed code and classifies each function by their compute-intensiveness. Once a candidate is found, it is first checked if it can be parallized and/or optimized using the Polly tool [86]. After all these steps are completed, the next time the candidate function is called during the execution, it will be off-loaded to the GPU. The HPA framework will also monitor the off-loaded functions and can revert its decision if their performance on the GPU proves to be lower than what the CPU was capable of.

Heterogeneous system with many accelerator types

The Heterogeneous Systems Architecture (HSA) Foundation was formed in June 2012 to enable the industry specification, advancement, and promotion of new chip designs, programming and runtime tools targeted for heterogeneous systems [8, 116]. The foundation members are among the largest tech companies today: AMD, ARM, Samsung, Texas Instruments and many others. They promote a lower level programming specification that serves as an intermediate layer definition between high-level language APIs (like OpenCL 2.0) and the underlying computing device [140]. HSA has so far released a specification for system architecture, an intermediate programming language called HSAIL, a runtime API and a multi-vendor architecture specification [9]. The end-goal is to achieve an unified install-base support across many devices and platforms which will enable software development in a *write once, run everywhere* fashion.

Early efforts of the HSA Foundation were on using the GPU as an accelerator (in HSA terminology they are referred to as *kernel agents* - see Figure 2.4 an example design), but its scope is to efficiently support a wide assortment of processing units. A HSA-compliant system should be able support multiple instruction sets based on host CPUs and several different kernel agents. It should also meet the requirements for an HSA queuing model, memory model and an instruction set for parallel processing. The HSA queuing model is designed to reduce the overhead of job dispatch by supporting shared virtual memory, system coherency, signaling and user mode queuing. The goal of the memory model is to ensure memory consistency in a concurrent parallel execution. Even though there are no GPUs to fully support the HSA specifications at the moment, early research and benchmarking shows that it brings important reduction in management overheads, increases performance and reduces latency [140].

Heterogeneous systems in HPC

Started in October 2011, the Mont Blanc European Union project aims to use embedded technology to develop the next generation HPC cluster capable of setting future global standards in the field [156, 157, 163]. With today's HPC systems, servers or data centers being power constrained, the Mont Blanc project is exploring a new approach in designing these systems in order to reach the exascale performance mark by 2018 [156]. The proposed prototype will use 15 compute cards which will include the ARM CPUs, GPUs, the DDR3 RAM memory and the Ethernet network switch in a single cluster blade [158]. Early results on a prototype cluster using NVIDIA Tegra 2 SoCs show lower power requirements than Intel x86 based reference systems, but a at cost of lower performance [156]. However, testing on newer SoCs like the Tegra 3 which has a higher multi-core density and a higher clock frequency per core, improves previous results reducing the performance gap [155].

The current Mont Blanc prototype, which is also available to public users, is using a total of 1080 compute cards, each fitted an Exynos 5 Dual SoC (ARM A15 dual-core and an

ARM Mali T-604 GPU). Grasso et al. have investigated the energy-efficiency potential of using this SoC for HPC workloads [85]. They conclude that embedded GPUs offer performance improvements of up to 8.7x and energy savings of nearly 3x on specific applications. They also identified some OpenCL software optimization techniques targeted for the ARM Mali GPU Compute Architecture. Speculative results of the prototype cluster using Exynos 5420 octa-core SoCs show it will surpass the performance of an Intel x86 Quad i7 reference system [158].

To help achieve the exascale performance goal giving the energy and hardware constraints, Keiser et all. propose a new a runtime system called HPX [105]. The authors argue most of the current HPC programming models are facing issues of scalability, programmability, performance portability, fault management and energy efficiency. HPX combines a set of well-known ideas and techniques like constraint based synchronization, adaptive locality control and message driven computation to reduces the challenges of executing on HPC architectures as they grow from peta- to exascale.

2.4 Programming model

Parallel programming is not a new topic and over the decades many parallel programming models have been developed and improved. A parallel programming model describes an abstract parallel machine by its basic operations such as arithmetic operations, spawning of tasks, reading from and writing to shared memory. Also, it outlines the effects of these operations on the state of the computation and the constraints of when and where these can be applied [108]. Such a parallel programming model is often associated with one or several parallel programming languages or libraries that realize the model.

In contrast with sequential programming, parallel programming models are not always platform agnostic. This is due to the fact that for parallel architectures there is not one unifying model. Leslie G. Valiant has credited the wide adoption of the sequential computation model throughout the ’70s and the ’80s to the fact that it adopted the von Neumann model. Foreseeing that general purpose parallel computation will some day replace the sequential one, he developed the initial Bulk Synchronous Parallel model (BSP) [170]. This model has since been improved and extended primarily with the help of Bill McColl and Rob Bisseling and their research teams [42, 43, 92, 134]. By 2010, Valiant has developed *Multi-BSP*, an extension to the original BSP model that better applies to multi-chip architectures [171]. A BSP system consists of processing units with their local memory, a network that can connect pairs of these processing units and a hardware mechanisms that allows the synchronization of all or a subset of components. Parallel computation is performed in a series of *supersteps* which includes concurrent execution of several threads, data communication and barrier synchronization.

In our research we focused on the *Task Based Programming* model (TBP). TBP, also known as nested task parallelism, promotes the parallelization by dividing a workload into small parts called *tasks*. In contrasts with *data parallelism*, the TBP tasks do not

have to include the same operations on different data. They can be different from each other and they can depend one on the other by passing data among them. When compared with the threads of *thread parallelism*, the TBP tasks are considerable lighter and platform agnostic. To increase flexibility and scalability of the model, the programmer marks the tasks to be executed in parallel, but it is the runtime system that actually manages the execution of the tasks. In this way, the same code can be executed on single-core, homogeneous or heterogeneous multi-core systems.

Tasks may spawn other tasks in a fork-join style and this may be done even in a dynamic and data dependent manner. Dependencies among tasks can either be specified by the programmer [76] or inferred automatically based on annotation of procedure declarations [67, 149]. Such collections of tasks may be represented by a task graph, where nodes represent tasks and arcs represent data dependencies, and can occur at several levels of granularity. At runtime, tasks are ordered and distributed automatically to available processing units and all dependencies are enforced. This approach frees the programmers from managing and mapping the parallel threads onto the CMP.

In the next subsections we present a few TBP models that we researched during our doctoral work. We did not used all of them in our papers (see Chapter 4), but discussing them here provides an overview of the state of the field.

2.4.1 OmpSs

The OmpSs programming model has been developed with the goal of supporting both homogeneous and heterogeneous hardware architectures [31, 67]. It extends OpenMP [52] with support for irregular and asynchronous parallelism and heterogeneous architectures from the StarSs programming model developed at the Barcelona Supercomputing Center (BSC)[149]. OmpSs uses the *in*, *out* and *inout* clauses to allow tasks to express data-dependencies. This information is used by the OmpSs runtime to create a dependency graph to control the order of execution and avoid explicit synchronization [58, 77]. In terms of heterogeneity, OmpSs is designed to simplify synchronization and data transfers between host CPUs and GPU accelerators. It supports the ability to schedule tasks to multiple GPUs independently and the possibility to provide more than one implementation of a task (in CUDA or OpenCL). The scheduler can select at runtime the task implementation based on resource availability and/or data locality [77].

Nanos++ is the run-time library developed at BSC [16] for most of the research-oriented task-centric models, such as OmpSs , StarSs [30] and SmpSs [130]. It relies on the Mercurium compiler [15] to translate the OmpSs constructs and transforms them into calls to the library [77]. Mercurium also parses the CUDA and OpenCL code that may be part of the source files, but it leaves this code unchanged. This gets passed to the NVIDIA compiler or the OpenCL runtime in the final steps of the compilation. Nanos++ is highly parametrized with options including different scheduling policies, cut-offs and back-off mechanisms for locks [77].

2.4.2 Intel Array Building Blocks

Intel Array Building Blocks (ArBB) is a high-level data-parallel programming environment. It is designed to produce scalable and portable programs that can harvest data and thread parallelism on both multi-core and heterogeneous many-core architectures [142]. Intel ArBB is a combination of Intel Ct, a former research project Intel started in 2007, and a multi-core development platform initially developed at the University of Waterloo [19] and then continued by RapidMind Inc.[18]. The aim of Intel ArBB is to increase productivity for programmers who need to exploit hardware parallelism, but for whom very low level parallel programming is not an option. Intel ArBB is an embedded language and is implemented as a library in the host language C++.

Bo Joel Svensson and Mary Sheeran worked towards embedding Intel ArBB into the Haskell functional programming language [164]. Their work extends a previous research that aims to accelerate Haskell array codes using GPUs [51]. Despite such promising results, Intel discontinued the ArBB development in favor of other projects like Cilk Plus and Threading Building Blocks.

2.4.3 Intel Cilk Plus

Intel Cilk Plus is a general-purpose programming language designed for multithreaded parallel computing and based on the C and C++ programming languages [20]. It extends these two languages to express task and data parallelism. Intel Cilk Plus allows parallelization of both new or existing code to efficiently exploit CMPs and/or vector instructions.

By marking a function call with the keyword *spawn*, the programmer indicates that it is safe to run that call in parallel. However, based on resource availability the scheduler will do the task management and assign them to the execution threads. Cilk also defines a barrier with the keyword *sync*, which signals that the current function call can't proceed until all previous spawned functions have completed. These two keywords coupled with parallel loops marked by *cilk_for* are the building blocks that provide software developers a very straightforward way to write parallel code [29].

In addition to good utilization of CMP resources, a central goal in the design of Intel products like Cilk Plus and Threading Building Blocks is *composability*. This feature has to do with the good collaboration among the components of an application that can be either custom designed or imported from third parties like libraries. This is achieved by the use of work-stealing task schedulers that allow the units of work to migrate from the execution thread that generated them to another one that will execute them. Such mechanism also ensures that the execution will exploit all the processing units available at any given time. These schedulers can accommodate to changes in the number or priority of applications executed on the system and scale down or up the number of execution threads accordingly.

2.4.4 Wool

Wool is a C-library supporting fine grained independent task parallelism and was developed by Karl-Filip Faxén at the Swedish Institute of Computer Science. Its main goals are to provide a simple programming interface and to address the very high management overheads that are common in fine-grained task parallelism. To achieve this Wool adopts a direct coding style to facilitate the parallelization of existing code. It also introduces the *direct task stack* approach to reduce overheads for basic operations like task spawning, joining and work stealing [74, 76].

Management overheads are an inherent challenge of parallelization. This becomes an even greater performance issue when working with fine-grained task parallelism. To address it, some TBP libraries implement cut-off thresholds to prevent the creation of finer tasks. In [61], the authors point out that when using TBB, tasks with less than 100k CPU cycles worth of work will incur a high performance penalty due to the overheads. In contrast, Wool is designed to handle much finer tasks and can achieve overheads twice as low when compared to other TBP implementations in some tests [76].

2.4.5 Intel Threading Building Blocks

Intel Threading Building Blocks (TBB) is a C++ template library designed to help software developers enable parallelism and scalability when writing loop- and task-based applications. TBB includes a number of generic parallel algorithms, a work stealing task scheduler, locks and atomic operations, highly concurrent containers and two memory allocator templates that address the issue of scalability. Like Intel Cilk Plus, TBB aims to decouple the programming from the particularities of the underlying machine [150].

To create a composable execution environment, the TBB task scheduler uses a thread pool that is visible at the process level. In this way it is capable of load-balancing the execution and quickly adapt to changes in resource availability. Building on top of the task scheduler, the library implements a number of highly tuned parallel algorithms (like *parallel_for* and *parallel_reduce*). To help developers manage their application's data TBB defines several concurrent containers (like *concurrent_queue* and *concurrent_vector*) and useful, low-level synchronization constructs (like *fetch_and_add* and *fetch_and_increment*). All these components provide an abstraction for parallelism that avoids the low level programming inherent in the direct use of threading packages such as p-threads.

2.4.6 What we used and why

Choosing among the various TBP implementations is tightly connected to the exact requirements of the application to be parallelized [12]. In the initial stages of the PhD

research, we needed a solution that will allow for a fast and easy parallelization of existing code. We were aiming to use in our research a benchmark suite with a wide user base against which we could compare our results. Both BOTS and PARSEC are parallel benchmark suits and come with one or several parallel implementations of their applications (see Section 3.2). We decided against compiler dependent solutions like OmpSs or Intel Cilk Plus since they are less flexible and a bit more difficult to work with. Because of its direct coding style and design goal of being low overhead, we first focused on Wool. The C-based library does not enjoy a large user base, but we were able to get support from its developer. Also we managed to do a collaboration with a team from KTH Royal Institute of Technology who had experience with Wool [98].

In a later stage of our research, we wanted to test the low overhead design of Wool against a solution which was not optimized for this [101]. Intel TBB was the best candidate for this, since it shared many characteristics with Wool (being a non-compiler dependent library with a work-stealing load balancing approach). Our work showed potential for reducing the overheads of Intel TBB, particularly those related to the stealing mechanism.

Chapter 3

Methodology

This chapter gives an overview of the experimental methodology used in our research. Based on previous work in our group, we developed a simulation framework to quantify our results on both performance and power efficiency. Section 3.1 presents our framework as a whole and then discusses each individual component. We then give a brief overview of the benchmarks used in our research in section 3.2.

3.1 Simulation framework

Developing or investigating in both hardware and software involves a significant effort for testing. On a high level, there are three main testing methodologies: analytical modeling, simulation and measurements on real hardware [161]. Analytical modeling has the advantage that it can efficiently cover large design spaces, but it can also be hard to reason about the accuracy of the findings. In contrast, measurements on real hardware can be very accurate but cover only one design point. Simulation bridges the gap between the other two by having the potential of good accuracy while covering reasonably large design spaces. It also has the advantage of providing a very large set of measurements that would otherwise be very difficult to attain.

Our research group has more than 10 years of experience in simulations based research in the multi-core area [68, 84, 104, 117]. For our research we needed a simulation framework that would allow us to measure both performance and energy-efficiency. The research of Marius Grannæs and Magnus Jahre [84, 104] recommended the M5 architectural simulator for performance measurements (see Section 3.1.1). At the start of our research, Wattch was the standard tool used in power and energy studies [45]. However, this tool was lacking the multithreaded and multi-core support we needed for our work. For this reason we opted for a newer tool called McPAT which had more features and was capable to integrate easily with our simulator (see Section 3.1.2). Figure 3.1 summarizes how the two tools interconnect and work together. Later on in our research we were able upgrade

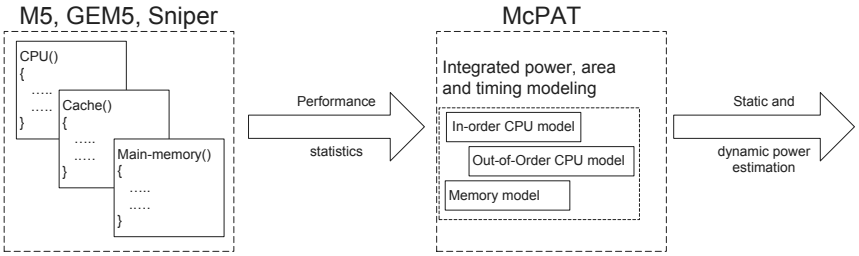


Figure 3.1: Simulation framework

our simulator, but maintain the workflow between it and the power estimation tool. Using this framework we were able to track down how changes in the software affected its execution which further allowed us to improve its performance and energy efficiency.

3.1.1 Architectural simulators

Over the course of our research we used a total of three different simulators: M5, GEM5 and Sniper. As already mentioned, we started our work using the execution-driven, full-system simulator called M5 [41]. The object oriented infrastructure of M5 reflects the modularity of real hardware. A simulated system is merely a collection of objects (CPUs, memory devices, interconnections etc.), all part of the same simulation process and sharing the same global event queue. This approach allowed us to focus only on the modules of interest to our research (core architecture, bus interconnect etc.) and abstract away the rest. M5 is able to do both *system call emulation* and *full-system simulation*. A full-system simulation includes the Operating System (OS) in addition to the test application/benchmark. In contrast, all system calls are handled by the host OS when doing system call emulation. In order to achieve a more realistic and deterministic simulation, we used the full-system approach in all of our work.

By incorporating the memory models of another simulator called GEMS [132], M5 evolved into GEM5 [40]. GEM5 maintained and improved all the main features and objects of M5 (CPU models, ISA support, I/O devices, infrastructure), but also added the cache coherence protocols and interconnect models of GEMS. This allowed users a wider range of investigations by using both the simple and fast memory model of M5, but also the detailed Ruby model from GEMS.

In the initial stage of our research (see Chapter 4) we pointed out an important limitation that is true for both M5 and GEM5: long simulation times. Consequently, this has prevented us from using anything but very small input sets when simulating with these two

tools. We dedicated part of our research to try to improve our framework and overcome this limitation (see Section 4.2). However, our approach to address this issue required more time for investigation and validation than we had available. At the same time a parallel computer architecture simulator called Sniper became available [49]. Sniper uses the interval core model [80] and Graphite simulation infrastructure [136] to provide fast and accurate simulations. With the ability to distribute the simulation across all available processing units in a host system, Sniper solved the problem we had with long simulation times. Moreover, Sniper was able to simulate the x86 ISA. This made it possible for us to execute Intel based parallelization libraries (like TBB) out-of-the-box without the need of cross-compilation or customization. On the other hand, by using Sniper we had to sacrifice some of the features available in GEM5, like the deterministic simulation.

All three simulators were easily integrated with McPAT by using a Python script to map the performance results to the input file of the power estimation tool (see Figure 3.1). As the tools in our framework changed or evolved, this "interconnect" had to be updated to ensure the correctness of the measurements.

3.1.2 Power estimation

The *Multicore Power, Area, and Timing* (McPAT) tool is a modeling framework that supports timing and space exploration for multi-core platforms [123]. It models all three types of power dissipation (dynamic, static, and short-circuit power) to give a complete view of the power usage of CMP processors. Developed in collaboration by HP-labs and the University of Notre Dame, McPAT models all major system components of a computer system (including in-order and out-of-order cores, network-on-chip, shared and private caches, memory controllers). McPAT enables architects to use metrics like energy-delay-area-squared product (EDA^2P) and energy-delay-area product (EDAP) that are useful to quantify the cost of new architectural ideas.

McPAT runs separately from the architectural simulator and only reads performance statistics from it through an XML-based interface. This interface allows both the specification of the static micro-architecture configuration parameters and the passing of dynamic activity statistics generated by the performance simulator. With this approach McPAT is very flexible and easily integrated with any number of simulators.

3.2 Benchmarks

In our first papers we used a subset of the *Barcelona OpenMP Task benchmark Suite* (BOTS) [66]. BOTS is a benchmark compilation assembled by the Barcelona Supercomputing Center (BSC) focused on assessing the performance of OpenMP task-based parallelization. The authors' aim was to both allow programmers to evaluate OpenMP

implementations on CMP architectures and investigate architectural effects on TBP applications. Some of the kernels were imported from other benchmark collections (like FFT or strassen) and others were written by the team at BSC (like sparselu). In all our experiments, we customized the benchmarks to use the Wool library rather than the default OpenMP parallelization.

To measure the parallelization overheads we first used a set of micro-benchmarks (Stress, MemStress and Matrix-Mul) written by the developer of the Wool library. These benchmarks were created to expose different features of a TBP library like spawning operations, load balancing the workload, and working with explicit or implicit parallelism. Together they cover both compute-intensive executions as well as memory-bounded ones.

In our last two papers we focused on a subset of the *Princeton Application Repository for Shared-Memory Computers* (PARSEC) benchmark suite [39]. Initially started as a cooperation between Intel and Princeton University, the development of PARSEC has expanded due to its open-source licensing. PARSEC is composed of multithreaded programs and was designed to be representative for the shared-memory workloads running on CMPs. It also provides an efficient manner to instrument and manipulate the included programs. The suite comes with a management script that facilitates the building of the complete suite or individual applications, selection of different building configurations, execution of the applications with different input sets and many other features. The suite includes applications from several domains (financial, computer vision, physical modeling, content based search etc.) and it was aimed at supporting research rather than being a scoring system [38]. A study by Bhadauria et al. has highlighted PARSEC's strong points in evaluating CMP architectural design, but it also emphasizes the limitation of most of its application to scale with the core count [35].

Since our study was focused on the overheads of Intel TBB, we selected the benchmarks that were parallelized using this library: Blackscholes, Bodytrack, Fluidanimate, Streamcluster and Swaptions. Collectively, these benchmarks express parallelism both explicitly as well as through some templates (like *parallel_for* and *parallel_reduce*). All these provide a good test base for our study.

Chapter 4

Research progress

This chapter discusses how we produced all the papers during my doctoral research. I illustrate the chronological order of the papers and how they are related. To simplify the discussion, the papers are grouped into three categories according to the main contribution of each paper. The categories are summarized in Table 4.1. Figure 4.1 shows the duration and concurrency of the work with the different papers as well as their relations.

Table 4.1 Papers categories

Category	Name	Number of papers
A	Literature review and initial study	5
B	Improving simulation framework	2
C	Investigating parallel overheads	3

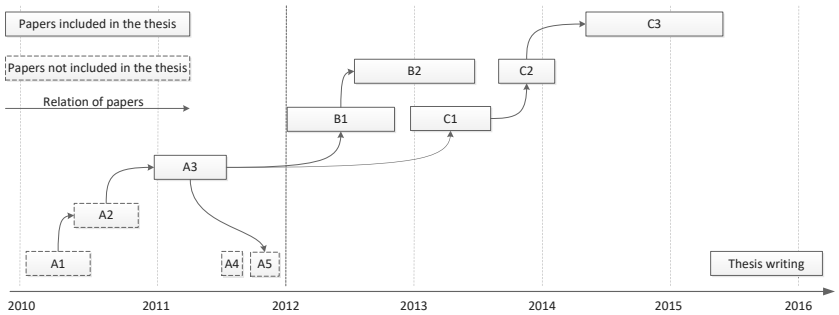


Figure 4.1: Papers chronology and logical connection.

4.1 Category A: Literature review and initial study

Table 4.2 Paper category A

	Title	Reference
A1	Energy Efficient Methods for Multi-core Programming	[96]
A2	Task Based Programming on Multicores, A Case Study on Energy Consumption	[97]
A3	Investigating the Potential of Energy-savings Using a Fine-grained Task Based Programming Model on Multi-cores	[98]
A4	Multi- and Many-cores, Architectural Overview for Programmers	[151]
A5	Green Computing: Saving Energy by Throttling, Simplicity and Parallelization	[141]

This PhD work was fully funded by the faculty of Information Technology, Mathematics and Electrical Engineering (IME) at NTNU. Part of the computation time needed for the experiments in this research was provided through the NOTUR project NN4650K *Multi-core Memory System Research* managed by the Department of Computer and Information Science (IDI), NTNU. With this project the CARD group at IDI was joining the *Green Computing* research trend, "a multifaceted global effort to reduce energy consumption and to promote sustainable development in the IT world" [115]. Concerns about increasing energy consumption managing costs and a general higher awareness about ICT's environmental impact, has pushed academia to find ways to improve energy efficiency and resource utilizations across all fields (see Figure 4.2). The approach I was tasked to study was improving the use of software parallelization to increase energy efficiency of CMPs.

The first step in my research was to do a literature study and put forward an initial research plan. Figure 4.2 summarizes my literature review, highlighting the areas of interest for my project. Table 4.2 includes all the papers we wrote during this initial stage of the PhD re-

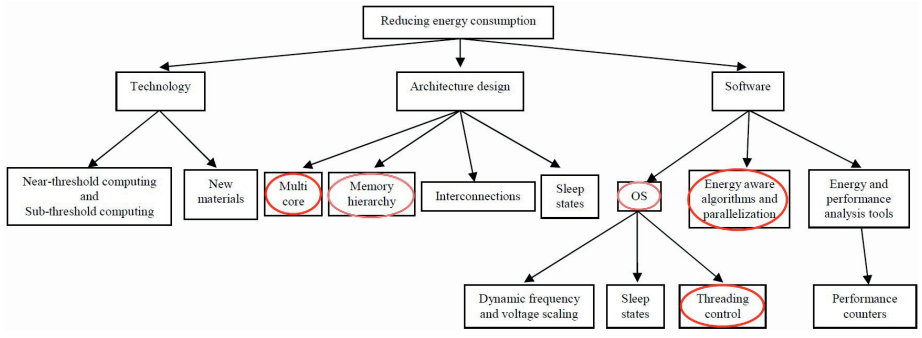


Figure 4.2: Overview of the literature review [96].

search. While attending the 6th summer school on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES 2010) I presented a poster discussing my research plan which was further detailed in paper A1.

This original plan was extended after the ACACES summer school, making task scheduling a center piece of our research. By this time I was able to do experiments using the first iteration of my simulation framework (M5 + McPAT). Also I already have chosen Wool as the TBP library to study and implemented some very simple applications to use as benchmarks. We set as goal for our research the *P6-goal*: Performant, Programmable, Portable and Power-efficient Parallel Programming. Paper A2 summarizes our research plan and our initial results designed to underline the energy efficiency gain of going from single-core to multi-core as well as the issue of parallelization overheads. I presented this paper during the third Swedish Workshop on Multi-Core Computing (MCC'10) in Göteborg, Sweden.

While attending MCC'10 I met Artur Podobas and Prof. Mats Brorsson from KTH Royal Institute of Technology. Artur was also at the start of his PhD research and he was focusing on quality-of-service improvements in task-parallel runtime-systems [152]. In early 2011 we discussed and agreed to work together for writing a paper investigating the energy savings that can be achieved by using TBP on CMPs. Artur had previously worked with Wool and he already ported the applications of the BOTS benchmark suite to use Wool rather than OpenMP. He provided these benchmarks which I used in my simulation framework to quantify the effects of parallelization on performance and energy consumption. Our results show improvements of the EDP metric if the parallel workload is properly balanced. With help and support from both our supervisors, this became paper A3 which I presented during the 2nd Workshop on Applications for Multi and Many Core Processors (A4MMC), part of ISCA 2011 in San Jose, California.

While working on paper A3, Prof. Lasse Natvig was invited to write an introductory chapter for a parallel programming book to be published by Wiley. To improve the understanding of the hardware-software synergy, this chapter was to give programmers an introduction into the history and architecture of CMPs. Together with another PhD candidate at that time, Mujahed Eleyat and two associate professors, Magnus Jahre and Jørn Amundsen, we wrote the *Multi- and Many-cores, Architectural Overview for Programmers* chapter. The chapter focuses on fundamental techniques (block access, functional parallelism, pipelining etc.), the architectural issues that lead to CMPs (the power wall, the memory wall etc.) and homogeneous/heterogeneous architectures with real world examples. The book entitled *Programming multi-core and many-core computing systems* is expected to be published in 2017.

In early 2011, Prof. Lasse Natvig learned about a special issue of the CEPIS Upgrade journal that would focus on Green Computing. Given that our research focus aligned with the topic of this journal, we agreed to write and submit a proposal. The aim of this article was to give an overview of several techniques that have been used for improving energy efficiency of computing systems. We emphasized parallelization and its role in both hardware and software to reduce energy consumption. After acceptance, this became

paper A5 which included work from our A1, A3 and A4 papers. The paper was also translated and published in the Spanish version of the journal.

4.2 Category B: Improving the simulation framework

Table 4.3 Paper category B

	Title	Reference
B1	Towards Efficient Simulation of Task Based Parallel Applications	[99]
B2	Challenges of reducing cycle-accurate simulation time for TBP applications	[100]

While working on paper A3, it became apparent that the full-system simulations we were performing with M5 suffered a very high time penalty. Applications that would execute in less than 2 seconds on real hardware could require up to 2 days to simulate in our framework. This meant that we could only use very small input sets in which the sequential portion of the execution would have a significant impact. This affected the granularity at which we could asses and quantify any changes we would do to the parallel code.

As part of my PhD training, I followed a course called *Computer Architecture 2* for which I also worked on a semester project. For this project I started investigating sample based techniques for reducing long simulation times. These techniques were successfully used for reducing simulation time of single-thread applications [89, 148, 182, 185], but there was no similar approach for parallel applications. The first issue to overcome when using sampling for simulated multi-threaded applications is to identify a metric that will allow you to correctly profile the execution. IPC is a poor candidate in this situation and a work related metric is needed [25]. For this reason I proposed the number of completed tasks as a progress metric. When looking at architecture features, TBP applications demonstrate a cyclic behavior across several metrics, including workload/time, branch prediction and cache performance. If correctly identified, this repeatable pattern can be used to reduce the simulation time. The method I proposed in the mini-project was a three stage process. First, the test application was simulated in a simple and fast simulation mode so it can be sampled and profiled. Second, on the resulting data set, I used the K-means clustering algorithm [128] to identify the repetitive sections in each execution. One representative sample from each cluster was then selected based on the cluster's centroid. Finally, I fast-forward the test application to each selected samples and simulate the samples in detailed mode. The results are weighted according to the size of the cluster they are part of and added up to create an overview of the complete execution. I observed speed-ups of up to 15x, but several issues concerning both the sampling approach and the M5/GEM5 simulator itself needed to be addressed.

As a first step in our work to improve the simulation framework, in paper B1 we focused on the sampling, clustering and representative points selection methods. For each of these

methods, we investigated several parameters and traced their impact on the accuracy and the speed of the sampled simulation. I customized the Wool library to signal and record every completed task. This was needed in order to use the number of completed tasks as a progress metric for sampling the TBP applications. Our results showed that an average of 5 samples, about 0.5 % of the total application, are sufficient to achieve an accuracy below 5 %. I presented these results in the Norsk Informatikkonferanse (NIK 2012), in Bødo, Norway.

Building on the results of the B1 paper, we developed a 3-phase methodology called FASTA to employ sampling for reducing simulations times of TBP applications. FASTA samples and profiles the execution, identifies representative sample points through clustering and simulates in detail the representative points. To improve my original approach from my semester project, I modified the GEM5 simulator to allow switching back-and-forward between simple and detailed simulation modes. We observed that the way the multiple execution threads interleave in a test execution differs from simulation to simulation and this is skewing the accuracy of FASTA. However, we recorde good accuracy (below 4%) a class of the applications we tested with a speedup of up to 12x. All these results are part of paper B2, which I presented in the International Conference on Computational Science (ICCS 2013), in Barcelona, Spain.

4.3 Category C: Investigating the parallel overheads

Table 4.4 Paper category C

	Title	**Reference**
C1	On the Energy Footprint of Task Based Parallel Applications	[101]
C2	Victim Selection Policies for Intel TBB: Overheads and Energy Footprint	[102]
C3	Tuning the victim selection policy of Intel TBB	[103]

While we were working on paper B2, Prof. Natvig learned about the release of a new parallel simulator called Sniper (see Section 3.1.1). Based on an earlier simulation infrastructure called Graphite developed at MIT [136] and using a new core model [80], Sniper was able to run a simulation in parallel. This feature reduced the simulation time and allowed us to use larger input sets for our test benchmarks. After investigating this new simulator for a while, we decided to use it rather than dedicating more time to the FASTA research. Building on the results of paper A3, we started looking into the parallelization overheads caused by some basic TBP operations like task spawning, task synchronization and task stealing. We decided to put to test Wool's low overhead design against Intel TBB. Using some synthetic benchmarks, we quantified the energy footprint for each of the above mentioned TBP operations across core counts ranging from 1 to 12 cores. Our results showed that even though Wool added far less overhead per TBP operation, its aggressiveness in balancing the workload by task stealing meant that it wasted more energy

overall. The results also showed that Intel TBB could also be improved to reduce the energy footprint of the parallelization overheads. I presented these results as paper C1 at the International Conference on High Performance Computing and Simulation (HPCS 13), in Helsinki, Finland.

Continuing the investigation of parallelization overheads and their energy footprint, we focused on the task stealing operation of Intel TBB. For testing, we decided to use a subset of the PARSEC suite, namely the Blackscholes, Bodytrack, Fluidanimate, Streamcluster and Swaptions benchmarks. Based on our results from the C1 paper, we focused on reducing the number of failed steal attempts. We observed that this type of overheads increases with core count which makes it an important limitation when scaling up the number of cores. To have a baseline of how much we can improve the victim selection mechanism, we first developed an *oracle scheme* that would always select a low congestion victim with some work to steal. To achieve this, we stored information about occupancy and congestion of the tasks queues outside the simulated memory space in our simulator. Updates to our data structure were done during the execution of the application through specialized instructions called *markers*. In addition, we also implemented a pseudo-random selection scheme inspired by Wool. This scheme will first select a random task queue to steal from and if that fails it then proceeds to scan all the other queues in an attempt to find work. Both of these schemes were compared against Intel TBB's random selection approach for both total number of instructions added as well as energy footprint. I presented our results as paper C2 at the International Conference on Architecture of Computing Systems (ARCS 2014), in Lübeck, Germany.

Latter the same year, we were given the opportunity to extend our C2 paper as an article to be published in a special number of the Journal of System Architecture (JSA). For this article, we kept the oracle scheme, we improved the implementation of the pseudo-random scheme and we introduced a new occupancy-aware selection scheme. We performed a more thorough comparison of the 3 schemes in terms of added overheads and we extended the maximum core count of the simulations from 16- to 32-cores. The article was published in JSA number 10, 2015.

Chapter 5

Research results

The aim of this chapter is to provide an overview of the papers included in this thesis. The sections 5.1 through 5.6 contain the abstract of the paper and a description of the contributions each co-author had to the paper.

5.1 Paper A3

Investigating the Potential of Energy-savings Using a Fine-grained Task Based Programming Model on Multi-cores

Alexandru C. Iordan, Artur Podobas, Lasse Natvig and Mats Brorsson
Presented at the 2nd Workshop on Applications for Multi and Many Core Processors
2011

5.1.1 Abstract

In this paper we study the relation between energy-efficiency and parallel executions when implemented with a fine-grained task-centric programming model. Using a simulation framework comprised of an architectural simulator and a power and area estimation tool, we have investigated the potential energy-savings when employing parallelism on multi-cores system. In our experiments with 2 - 8 multi-cores systems, we employed frequency and voltage scaling in order to keep the relative performance of the systems constant and measured the energy-efficiency using the *Energy-delay-product*. Also, we compared the energy consumption of the parallel execution against the serial one. Our results show that through judicious choice of load balancing parameters, significant improvements of around 200 % in energy consumption can be acheived.

5.1.2 Authors' contribution

The work in this paper continues the experiments we started in paper A2. It was Prof. Natvig's proposal to do a comparative study of the energy consumption for single-core execution against several multi-core parallel ones. I developed the experimental framework which involved installing the M5 simulator and McPAT tool on the Kongull cluster computer at NTNU, cross-compiling all benchmarks and improving the Python script used to transfer the performance results from M5 simulation to the input file of McPAT. I also performed all the experiments and wrote most of the text of the paper. Artur Podobas provided the BOTS benchmarks that he ported to use the Wool library as well as a brief description for each of them. He and I had several discussions over e-mail or Skype over methodology, how to present the results and changes to the body of the text. Both Prof. Natvig and Prof. Brorsson worked as advisors providing many helpful comments and improvements to the overall quality of the paper.

5.2 Paper B1

Towards Efficient Simulation of Task Based Parallel Applications

Alexandru C. Iordan, Magnus Jahre and Lasse Natvig

Norsk Informatikkonferanse (NIK)
2012

5.2.1 Abstract

For computer architects and software developers, simulation is an indispensable tool for evaluating new designs and ideas. Unfortunately, it is significantly slower to simulate a parallel environment than to work on real hardware. In this work, we take a first step towards efficient simulation of Task Based Parallel (TBP) applications. Our key idea is that the number of completed tasks can be used as a work-related progress metric to sample these applications. Using this metric, we show that the complete execution of TBP programs can be accurately represented by simulating only a few samples. In fact, our TBP applications only need 5 samples on average to get a mean error below 5 %. In our experiments, when sampling at 1000 completed tasks, 5 samples correspond to 0.5 % of the total execution on average.

5.2.2 Authors' contribution

Inspired by the sampling approach of SimPoint [89], I came up with the idea for this paper while working on the *Computer Architecture 2* semester project. I extended the experi-

mental framework, which involved customizing the Wool library to signal GEM5 when a task was completed, customizing GEM5 to quantify the total number of completed tasks and trigger a statistic dump for each sample size and various scripting for extracting information from the samples. I also performed all the experiments and wrote most of the text of the paper. Assoc. prof. Magnus Jahre provided much support throughout the development of this work, both in terms of the technical work as well as improvements to the text. Prof. Natvig worked as advisor providing many helpful comments and improvements to the overall quality of the paper.

5.3 Paper B2

Challenges of Reducing Cycle-accurate Simulation Time for TBP Applications

Alexandru C. Iordan, Magnus Jahre and Lasse Natvig

International Conference on Computational Science (ICCS)

2013

5.3.1 Abstract

Cycle-accurate simulation is an important tool that depends on the computational power of supercomputers. Unfortunately, simulations of modern multi-core platforms can take weeks or months. In this paper, we look into the challenges of employing a sampling based technique for reducing simulation time of multi-threaded applications. We introduce FASTA, a simple 3-phase methodology for reducing the simulation time of Task Based Parallel applications. FASTA takes advantage of the periodic behavior of parallel applications and identifies a small number of representative execution samples. By exploring a large design space we show that even though we can not use FASTA for every type of application, there are some for which a 12x speedup can be achieved with an accuracy error as low as 2.6%.

5.3.2 Authors' contribution

This paper is a continuation of the work I did for the semester project and the B1 paper. I extended the experimental framework, which involved customizing the GEM5 simulator to allow the back-and-forward switching between simple and detailed simulation modes and to allow multiple checkpoints to be created. I also performed all the experiments and wrote most of the text of the paper. Assoc. prof. Magnus Jahre provided much support throughout the development of this work. His help needs to be primarily noted in the customization of the GEM5 simulator to allow the back-and-forward switching between

simple and detailed simulation modes. Prof. Natvig worked as advisor providing many helpful comments and improvements to the overall quality of the paper.

5.4 Paper C1

On the Energy Footprint of Task Based Parallel Applications

Alexandru C. Iordan, Magnus Jahre and Lasse Natvig

International Conference on High Performance Computing and Simulation (HPCS)

2013

5.4.1 Abstract

From HPC systems to embedded devices, energy consumption is becoming a dominant factor in managing costs. With Chip multiprocessors becoming the platform of choice in almost all ICT segments, software developers need to employ parallel programming to fully exploit this architecture. However, parallelization adds a management overhead to the execution of an application. In this paper, we study parallel applications implemented using two TBP libraries, Wool and Intel TBB. We explore both compute and memory bound executions. We also investigate the energy footprint of the parallelization overhead and the effect it has on the energy-efficiency of the executing system. Our study looks into the behavior of some basic parallelization operations like task spawning, task synchronization and task stealing. We encountered situations when the number of task stealing operations grows exponentially with the core count increasing execution time. This behavior drastically reduces the energy-efficiency of those executions. Avoiding such behavior is crucial if future parallel systems are to reach their performance potential. Our results also show that the energy efficiency for compute intensive applications improves with the core count while for memory intensive application that is not the case.

5.4.2 Authors' contribution

The idea for this paper resulted from the conclusions of paper A3. Previous to this paper, Prof. Natvig suggested a new parallel simulator called Sniper that I tested and included in our experimental framework. I moved our experimental framework to the Stallo supercomputer at UiT, in Trømso and used IDI's computing time as part of the EECS research initiative. I customized the TBP libraries to signal the simulator when a task operation (spawn, sync and steal) has started and ended and I customized the Sniper simulator to dump its statistics at each signal. I created the scripts to parse all the resulting data and quantify for each task operation its effect on the execution time and energy efficiency of

the application. I performed all the experiments and wrote most of the text of the paper. Assoc. prof. Magnus Jahre and I had several discussions about the paper and he provided numerous helpful comments on methodology, results and text. Prof. Natvig worked as advisor providing feedback and improvements to the overall quality of the paper.

5.5 Paper C2

Victim Selection Policies for Intel TBB: Overheads and Energy Footprint

Alexandru C. Iordan, Magnus Jahre and Lasse Natvig
Architecture of Computing Systems (ARCS)
2014

5.5.1 Abstract

With the wide adoption of Chip Multiprocessors (CMPs), software developers need to switch to parallel programming to reach the performance potential of CMPs and maximize their energy efficiency. Management overheads due to parallelization can cause sub-linear speedups and increase the energy consumption of parallel programs. In this paper, we investigate the parallelization overheads of Intel TBB with a particular focus on its victim selection policy. We implement an "all knowing" oracle victim selection scheme as well as a pseudo-random scheme and compare them against TBB's default random selection policy. We also break down TBB's parallelization overheads and report how basic operations like task spawning, task stealing and task de-queuing impact the energy footprint. Our experiments show that failed task stealing is by far the highest energy consumer. In fact, the oracle victim selection policy can reduce the application energy footprint by 13.6% compared to TBB's default policy.

5.5.2 Authors' contribution

This paper continues the work in paper C1. I extended the experimental framework, which involved compiling the selected PARSEC benchmarks with our customized TBB library. I also performed all the experiments and wrote most of the text of the paper. Assoc. prof. Magnus Jahre provided much support throughout the development of this work on methodology, results and text. Prof. Natvig worked as advisor providing feedback and improvements to the overall quality of the paper.

5.6 Paper C3

Tuning the victim selection policy of Intel TBB

Alexandru C. Iordan, Magnus Jahre and Lasse Natvig
Journal of Systems Architecture (JSA)
2015

5.6.1 Abstract

The wide adoption of Chip Multiprocessors (CMPs) in almost all ICT segments has triggered a change in the way software needs to be developed. Parallel programming maximizes the performance and energy efficiency of CMPs, but also comes with a new set of challenges. Parallelization overheads can account for sub-linear speedups and can increase the energy consumption of applications. In past experiments we looked at specific operations such as spawning new tasks, dequeuing the task queue and task stealing for Intel TBB. Our results showed that failed steals account for the largest overhead. In this work, we focus on TBB's victim selection policy. We implement a new occupancy-aware policy and we improve the implementation of the pseudo-random policy we proposed in a previous paper. We compare the results of our new policies against an "oracle scheme" as well as against TBB's random victim selection approach. Our results show improvements in execution times and energy-efficiency of up to 11.23% and 14.72% respectively when compared to TBB's default policy.

5.6.2 Authors' contribution

This paper extends the work in paper C2. I performed all the experiments and wrote most of the text of the paper. Both assoc. prof. Magnus Jahre and Prof. Natvig provided much support throughout the development of this work on presenting the results and improvements to the overall quality of the paper. Jahre and Natvig's support needs a special mention for this paper, since they managed to keep me focused to deliver the paper on a tight and stressful schedule.

Chapter 6

Concluding remarks

The goal of this thesis was to show in what way parallel programming can be used to improve a system's power efficiency. We focused on the management of parallelization overheads and how they impact the energy footprint of an execution. We have investigated the overheads related with the basic TBP operations, like task spawning, task synchronization and task stealing. This investigation showed that task stealing does not scale with core counts and is responsible for performance bottlenecks. We proposed several victim selection schemes to reduce the number of failed steal attempts and improve both execution time and energy efficiency of the application.

We started in paper A3 by showing how parallel executions can very easily reduce the energy consumption when compared to sequential ones. While increasing the number of processing cores, but keeping the relative performance of the system constant, our results showed a *bathtub* trend in energy consumption. This trend underlines the effect parallelization overheads have on the execution. Continuing the investigation in paper C1, we looked at basic TBP operations and quantified their effect on the energy footprint of our test applications. We singled out task stealing as the main bottleneck among tested overheads and we designated it as main area of interest for moving forward. In papers C2 and C3 we focused on reducing the number of failed steal attempts and improving both execution times and energy consumption by changing the victim selection policy. Our results show that an informed victim selection can be very beneficial and can eliminate the performance degradation we observed while increasing the core count.

In addition we also performed work towards reducing simulation times of deterministic architectural simulators like M5/GEM5 in papers B1 and B2. Our work was inspired by sampling methodologies like SimPoint used in single-core simulations [89]. We proposed the number of completed tasks to be used as a work-related progress metric to sample simulated parallel applications. We introduced FASTA, a 3-phase methodology that employs sampling to reduce simulations times of TBP applications.

6.1 Contributions

Our main goal in this research was to investigate how parallel programming can be used to improve the energy efficiency of applications running on CMP systems. We further divided this goal in three research questions. In the next sections I will try to review these questions considering the results included in the thesis.

6.1.1 Research question 1

RQ1: What is the potential for energy saving for the TBP model on a multi-core system?

Paper A2 includes the initial results, but the main work was done in paper A3. By parallelizing a set of benchmarks and comparing their EDP for single-core execution against the parallel executions across several core counts we could see both the benefits and trade-offs of parallelization. The *bathtub* trend reflects very well how parallelizations improves a system's efficiency as well as how management overheads can overcome the benefits and push the EDP metric high. All benchmarks recorded their lowest EDP for core counts between 2 and 4 after which we can see how execution times increase, driving the EDP up. The reason we are seeing this effect at such low core counts is due to the very small input sets we were using at that time. However, the overall results gave us enough data to continue our research further, as well as opened new challenges for us which we tried to tackle in papers B1 and B2.

6.1.2 Research question 2

RQ2: How do basic operations in TBP parallel code (task spawning, task migration, synchronization etc.) affect the energy footprint of the execution? What are the performance/energy trade-offs?

In paper C1 we did a comparative study of Wool and Intel TBB in an attempt to identify best working scenarios for both TBP libraries and highlight their strong points. By design, Wool is a lightweight, low overhead library and as we expected outperformed Intel TBB in both execution time and energy efficiency. However, a closer look at some basic TBP operations showed that Wool is more aggressive in trying to balance the workload and the energy footprint of task stealing increases exponentially because of that. For operations like task spawning and task synchronization, both libraries show almost constant energy footprint across core counts. This is due to the fact that these operations are influenced mainly by the size of the input set. In regard to the task stealing bottleneck, Intel TBB performs better in the sense that it includes cut-off mechanisms that put worker threads to sleep and reduce congestion when there is too little work. Because of the results in this

paper we focused our attention on the task stealing operation in an attempt to eliminate the performance bottleneck it introduces as you scale up the core count.

6.1.3 Research question 3

RQ3: How can we improve the energy efficiency of TPB programs?

Building on the results of paper C1, we wanted to improve the work stealing mechanism in Intel TBB, particularly to reduce the number of failed steal attempts. In papers C2 and C3 we proposed and evaluated several victim selection schemes. The default approach in Intel TBB is to use a random selection of victims and a hard coded threshold for how many failed attempts before putting a worker thread to sleep. With our new schemes we could put the worker threads to sleep faster when there was too little work to do. Our approach showed good results for some benchmarks, but not all of them. The extra work we added to the library could not always be balanced out and in those cases our implementations showed a higher energy footprint than the default one.

6.1.4 Extra research question

ERQ: How can we reduce long simulation times of deterministic architectural simulators like M5/GEM5?

Our results in paper A3 pointed out the long simulation times as very important limitation of our experimental framework. Because of this limitation, it was not feasible for us to use anything but very small input sets in our testing. In papers B1 and B2 we tried to find a way to address this issue and we developed a sampling based methodology called FASTA. FASTA could reduce simulation times by a factor of 12x, but it would also reduce the accuracy of the simulation. Since this was not the main goal of our research, when a new parallel architectural simulator became available we decided to update our experimental framework rather than continuing refining FASTA.

6.2 Future work

The simulation framework we employed in our research provides extensive performance results and power estimations which would be very difficult to extract from a real world platform. However, the accuracy of these results can vary and for this reason we focused on the trend of our results rather than raw values. To evaluate more accurately the impact of the parallelization overheads on a given CMP architecture, real world executions and measurements can be performed. Juan M. Cebrián et al. used the Model Specific Registers (MSRs) present in Intel's Core architecture since the second generation (Sandy

Bridge) to measure the chip's energy efficiency [50]. Their results also reveal that temperature variation can have an important impact on energy consumption, something that our framework could not simulate.

Our study of the overheads involved in Intel TBB's task stealing mechanism revealed two important points. First, the very simple default approach of random victim selection does not scale well with core count. As the number of cores increases, randomly picking a victim to steal from fails to help with the congestion issue. An informed decision about which thread has work to share and/or the level of congestion of each thread can reduce the performance penalty observed with the random selection approach. However, gathering such information at runtime also incurs a performance penalty which cannot always be balanced out. This is our second important observation. As a future development, combining the two selection methods and adding a switching mechanism between them could provide better results on a wider range of scenarios.

In power constrained fields, like mobile, the CPUs are designed to trade performance for energy-efficiency by dynamically throttling down operating frequencies or even powering down some of the cores. Such mechanisms could also be applied for software design and application could be coded with the ability to switch between an *energy efficient* profile and a *performance* profile. Ideally the switch should be handled by the OS, but that will also require extending the existing APIs in the OS.

6.3 Outlook

Until new technologies will emerge to address the power wall, the memory wall and the ILP wall [88, 177, 181], CMPs are the architecture of choice in all fields of ICT. With CMPs becoming ubiquitous, parallel programming becomes a mandatory skill in order to fully exploit such architectures. Because of a large number of problem types, a single *silver bullet* hardware and/or software model is highly unlike to emerge. Instead, pushed by Amdahl's law and by the need for specialized computations, CMPs will become more heterogeneous [93]. Challenges like high power density will require CMPs to be able to dial down their operating frequencies or move execution to less power intensive cores [71]. With the big.LITTLE architecture, ARM's line of mobile CMPs has proven that high performing and energy efficient operating modes can coexist on the same chip.

On the software side, parallelization models and/or libraries will need to provide developers with an abstraction level that will make parallel programming more main stream. Code composability will need to become a main design goal for any application developed for multi-threaded execution. This is needed to allow runtime resource allocation for multiple parallel application running on the same CMP.

Bibliography

[1] ARM's Cortex A57 and Cortex A53: The First 64-bit ARMv8 CPU Cores. http://www.anandtech.com/show/6420/arms-cortex-a57-and-cortex-a53-the-first-64bit-armv8-cpu-cores. (retrieved November 2016).

[2] AMD Compute Libraries. http://developer.amd.com/tools-and-sdks/opencl-zone/acl-amd-compute-libraries/. (retrieved November 2016).

[3] Early Computers at Stanford. http://forum.stanford.edu/wiki/index.php/Early_Computers_at_Stanford#DEC_PDP-10_2. (retrieved November 2016).

[4] The Samsung Exynos 7420 Deep Dive - Inside a Modern 14nm SoC. http://www.anandtech.com/show/9330/exynos-7420-deep-dive, . (retrieved November 2016).

[5] ARM A53/A57/T760 investigated - Samsung Galaxy Note 4 Exynos Review. http://www.anandtech.com/show/8718/the-samsung-galaxy-note-4-exynos-review, . (retrieved November 2016).

[6] Up close and personal: how the Samsung Galaxy S6 uses its octa-core processor. http://www.androidauthority.com/galaxy-s6-octa-core-processor-usage-617585/, . (retrieved November 2016).

[7] The Green 500 list. http://www.green500.org/. (retrieved November 2016).

[8] What is Heterogeneous System Architecture (HSA)? http://www.hsafoundation.com/what-is-heterogeneous-system-architecture-hsa/, . (retrieved November 2016).

[9] HSA Standards to Bring About the Next Level of Innovation. http://www.hsafoundation.com/standards/, . (retrieved November 2016).

[10] International Technology Roadmap for Semiconductors 2007 Edition. http://www.itrs.net/links/2007itrs/ExecSum2007.pdf. (retrieved November 2016).

[11] International Technology Roadmap for Semiconductors 2007 Edition, The Model for Assessment of CMOS Technologies and Roadmaps (MASTAR). http://www.itrs.net/models.html. (retrieved November 2016).

[12] Choosing the right threading framework. https://software.intel.com/en-us/articles/choosing-the-right-threading-framework. (retrieved November 2016).

[13] Exynos 5 Octa (5422). http://www.samsung.com/semiconductor/minisite/Exynos/w/solution/mobile_ap/5422/. (retrieved November 2016).

[14] The M5 Simulator System webpage. http://www.m5sim.org/wiki/index.php/Main_Page. (retrieved November 2016).

[15] The Mercurium compiler. https://pm.bsc.es/mcxx. (retrieved November 2016).

[16] The Nanos++ runtime library. https://pm.bsc.es/nanox. (retrieved November 2016).

[17] OpenCL: Portable Heterogeneous Computing. https://www.khronos.org/assets/uploads/developers/library/overview/opencl_overview.pdf. (retrieved November 2016).

[18] RapidMind: C++ Meets Multicore. http://www.drdobbs.com/parallel/rapidmind-c-meets-multicore/199902702. (retrieved November 2016).

[19] Sh: A high-level metaprogramming language for modern GPUs. http://libsh.org/index.html. (retrieved November 2016).

[20] Intel Cilk Plus. https://software.intel.com/en-us/intel-cplusplus-compiler-16.0-user-and-reference-guide-cilk-plus, 2011. (retrieved November 2016).

[21] Personal Computing Transformed: Ultrabook Ushers In the Most Complete Computing Experience. http://newsroom.intel.com/community/intel_newsroom/blog/2011/09/14/personal-computing-transformed-ultrabook-ushers-in-the-most-complete-computing-experience, 2011. (retrieved November 2016).

[22] Intel Integrated Performance Primitives - User's Guide. https://software.intel.com/sites/default/files/managed/a7/15/ipp_userguide_0.pdf, 2011. (retrieved November 2016).

[23] Intel Math Kernel Library - Reference manual. https://software.intel.com/en-us/mkl-reference-manual-for-c, 2011. (retrieved November 2016).

[24] Intel Threading Building Blocks - Documentation. https://software.intel.com/en-us/tbb-documentation, 2011. (retrieved November 2016).

[25] A. R. Alameldeen and D. A. Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17, July 2006. ISSN 0272-1732. doi: 10.1109/MM.2006.73. URL http://dx.doi.org/10.1109/MM.2006.73.

[26] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, M. D. Hill, D. A. Wood, and D. J. Sorin. Simulating a $2M Commercial Server on a $2K PC. *Computer*, 36(2), 2003.

[27] G. M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM. doi: 10.1145/1465482.1465560. URL `http://doi.acm.org/10.1145/1465482.1465560`.

[28] M. Arora. The Architecture and Evolution of CPU-GPU Systems for General Purpose Computing, Research survey, Department of Computer Science and Engineering, UC San Diego. `http://cseweb.ucsd.edu/~marora/files/papers/REReport_ManishArora.pdf`, 2012. (retrieved November 2016).

[29] R. Asai and A. Vladimirov. Intel Cilk Plus for Complex Parallel Algorithms: "Enormous Fast Fourier Transform" (EFFT) Library. *CoRR*, abs/1409.5757, 2014. URL `http://arxiv.org/abs/1409.5757`.

[30] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03868-6. doi: 10.1007/978-3-642-03869-3_79. URL `http://dx.doi.org/10.1007/978-3-642-03869-3_79`.

[31] E. Ayguadé, R. M. Badia, P. Bellens, D. Cabrera, A. Duran, R. Ferrer, M. Gonzàlez, F. Igual, D. Jiménez-González, J. Labarta, L. Martinell, X. Martorell, R. Mayo, J. M. Pérez, J. Planas, and E. S. Quintana-Ortí. Extending OpenMP to Survive the Heterogeneous Multi-Core Era. *International Journal of Parallel Programming*, 38(5):440–459, 2010. ISSN 1573-7640. doi: 10.1007/s10766-010-0135-4.

[32] K. C. Barr, H. Pan, M. Zhang, and K. Asanovic. Accelerating Multiprocessor Simulation with a Memory Timestamp Record. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, pages 66–77, March 2005. doi: 10.1109/ISPASS.2005.1430560.

[33] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 282–293, June 2000.

[34] L. A. Barroso and U. Höelzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[35] M. Bhadauria, V. M. Weaver, and S. A. McKee. Understanding PARSEC Performance on Contemporary CMPs. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 98–107, Washington,

DC, USA, October 2009. IEEE Computer Society. ISBN 978-1-4244-5156-2. doi: 10.1109/IISWC.2009.5306793. URL http://dx.doi.org/10.1109/IISWC.2009.5306793.

[36] A. Bhattacharjee and M. Martonosi. Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors. In *Proc. of the 36th annual Int'l Symp. on Computer Architecture*, 2009.

[37] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[38] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pages 47–56, September 2008. doi: 10.1109/IISWC.2008.4636090.

[39] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, October 2008. ACM. ISBN 978-1-60558-282-5. doi: 10.1145/1454115.1454128. URL http://doi.acm.org/10.1145/1454115.1454128.

[40] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The GEM5 Simulator. *SIGARCH Comput. Archit. News*, 39, 2011.

[41] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. 26(4):52–60, July 2006. ISSN 0272-1732. doi: 10.1109/MM.2006.82. URL http://dx.doi.org/10.1109/MM.2006.82.

[42] R. H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*. Oxford University Press, 2004. ISBN 0198529392.

[43] R. H. Bisseling and W. F. McColl. Scientific Computing on Bulk Synchronous Parallel Architectures. In *Proceedings of 13th IFIP World Computer Congress*, volume 1, 1994. URL http://www.comlab.ox.ac.uk/oucl/users/bill.mccoll/p4.ps.Z.

[44] S. Borkar and A. A. Chien. The Future of Microprocessors. *Commun. ACM*, 54 (5), May 2011.

[45] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: A Framework for Architectural-level Power Analysis and Optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 83–94, New York, NY, USA, 2000. ACM. ISBN 1-58113-232-8. doi: 10.1145/339647.339657. URL http://doi.acm.org/10.1145/339647.339657.

[46] A. Butko, A. Gamatié, G. Sassatelli, L. Torres, and M. Robert. Design Exploration for next Generation High-Performance Manycore On-chip Systems: Application to big.LITTLE Architectures. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 551–556, July 2015. doi: 10.1109/ISVLSI.2015.28.

[47] A. Butko, L. Bessad, D. Novo, F. Bruguier, A. Gamatié, G. Sassatelli, L. Torres, and M. Robert. Position Paper: OpenMP scheduling on ARM big.LITTLE architecture. In *Ninth International Workshop on Programmability and Architectures for Heterogeneous Multicores (MULTIPROG-2016)*, Jan 2016.

[48] R. B. Cameron McNairy. Montecito - The next product in the Itanium(R) Processor Family. `http://www.hotchips.org/archive/`, 2004. (retrieved November 2016).

[49] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, Nov 2011.

[50] J. M. Cebrián, M. Jahre, and L. Natvig. Optimized hardware for suboptimal software: The case for SIMD-aware benchmarks. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 66–75, March 2014. doi: 10.1109/ISPASS.2014.6844462.

[51] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover. Accelerating Haskell Array Codes with Multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0486-3. doi: 10.1145/1926354.1926358. URL `http://doi.acm.org/10.1145/1926354.1926358`.

[52] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming*. MIT Press, 2007.

[53] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10): 1370 – 1380, 2008. ISSN 0743-7315. doi: {http://dx.doi.org/10.1016/j.jpdc.2008.05.014}. URL `http://www.sciencedirect.com/science/article/pii/S0743731508000932`.

[54] J. Chen, M. Annavaram, and M. Dubois. SlackSim: a platform for parallel simulations of CMPs on CMPs. *SIGARCH Comput. Archit. News*, 37, 2009.

[55] X. Chen, W. Chen, J. Li, Z. Zheng, L. Shen, and Z. Wang. Characterizing Fine-Grain Parallelism on Modern Multicore Platform. In *IEEE 17th Int'l Conf. on Parallel and Distributed Systems*, 2011.

[56] N. Chitlur, G. Srinivasa, S. Hahn, P. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prab-

hakaran, L. Zhao, N. Ijih, S. Subhaschandra, S. Grover, X. Jiang, and R. Iyer. QuickIA: Exploring heterogeneous architectures on real prototypes. In *IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–8, Feb 2012. doi: 10.1109/HPCA.2012.6169046.

[57] S. Cho and R. G. Melhem. On the interplay of parallelization, program performance, and energy consumption. *IEEE Transactions on Parallel and Distributed Systems*, 21(3):342 –353, 2010.

[58] K. Chronaki, A. Rico, R. M. Badia, E. Ayguadé, J. Labarta, and M. Valero. Criticality-Aware Dynamic Task Scheduling for Heterogeneous Architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 329–338, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3559-1. doi: 10.1145/2751205.2751235. URL http://doi.acm.org/10.1145/2751205.2751235.

[59] Cilk++. Cilk++: A quick, easy and reliable way to improve threaded performance. Available: http://software.intel.com/en-us/articles/intel-cilk-plus/. (retrieved November 2016).

[60] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing State Loss For Effective Trace Sampling of Superscalar Processors. In *Proc. of the International Conference on Computer Design, VLSI in Computers and Processors*, 1996.

[61] G. Contreras and M. Martonosi. Characterizing and Improving the Performance of Intel Threading Building Blocks. In *IEEE International Symposium on Workload Characterization*, pages 57–66, Sept 2008. doi: 10.1109/IISWC.2008.4636091.

[62] I. datasheet. Intel Pemtium M processor on 90 nm process with 2-MB L2 cache. http://download.intel.com/support/processors/mobile/pm/sb/30218908.pdf. (retrieved November 2016).

[63] B. Delporte, R. Rigamonti, and A. Dassatti. HPA: An Opportunistic Approach to Embedded Energy Efficiency. In *International Conference on High Performance Computing Simulation (HPCS)*, pages 792–799, July 2016. doi: 10.1109/HPCSim.2016.7568415.

[64] R. Dennard, V. Rideout, E. Bassous, and A. LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974. ISSN 0018-9200. doi: 10.1109/JSSC.1974.1050511.

[65] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing*, 38(8): 391 – 407, 2012. ISSN 0167-8191. doi: {http://dx.doi.org/10.1016/j.parco.2011.10.002}. URL http://www.sciencedirect.com/science/article/pii/S0167819111001335.

[66] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing*, pages 124–131, Sept 2009. doi: 10.1109/ICPP.2009.64.

[67] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21, 2011.

[68] H. Dybdahl. *Architectural Techniques to Improve Cache Utilization*. PhD thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2010.

[69] H. Dybdahl, P. Stenstrom, and L. Natvig. A Cache-Partitioning Aware Replacement Policy for Chip Multiprocessors. In Y. R. et al., editor, *High Performance Computing - HiPC 2006*, volume 4297 of *LNCS*, pages 22–34. Springer Berlin / Heidelberg, 2006.

[70] L. Eeckhout. Computer Architecture Performance Evaluation Methods. *Synthesis Lectures on Computer Architecture*, 5(1):1–145, 2010. doi: 10.2200/S00273ED1V01Y201006CAC010.

[71] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0472-6. doi: 10.1145/2000064.2000108. URL `http://doi.acm.org/10.1145/2000064.2000108`.

[72] S. Eyerman and L. Eeckhout. System-Level Performance Metrics for Multiprogram Workloads. *Micro, IEEE*, 28(3):42–53, May 2008. ISSN 0272-1732. doi: 10.1109/MM.2008.44.

[73] K.-F. Faxén. Wool 0.1 users guide. `http://www.sics.se/~kff/wool/users-guide.pdf`.

[74] K.-F. Faxén. Wool - A Work Stealing Library. *SIGARCH Comput. Archit. News*, 36(5):93–100, June 2009. ISSN 0163-5964. doi: 10.1145/1556444.1556457. URL `http://doi.acm.org/10.1145/1556444.1556457`.

[75] K.-F. Faxén. Efficient Work Stealing for Fine Grained Parallelism. In *39th Int'l Conf. on Parallel Processing*, 2010.

[76] K.-F. Faxén. Efficient Work Stealing for Fine Grained Parallelism. In *Proceedings of 39th International Conference on Parallel Processing*, San Diego, 2010.

[77] A. Fernández, V. Beltran, X. Martorell, R. M. Badia, E. Ayguadé, and J. Labarta. Task-Based Programming with OmpSs and Its Application. In *Euro-Par 2014: Parallel Processing Workshops: Euro-Par 2014 International Workshops*, pages 601–612. Springer International Publishing, 2014. ISBN 978-3-319-14313-2. doi:

10.1007/978-3-319-14313-2_51. URL http://dx.doi.org/10.1007/978-3-319-14313-2_51.

[78] S. Fuller and L. Millett. Computing Performance: Game Over or Next Level? *Computer*, 44(1):31–38, Jan 2011. ISSN 0018-9162. doi: 10.1109/MC.2011.15.

[79] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, May 2005. ISSN 0018-9162. doi: 10.1109/MC.2005.160.

[80] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval Simulation: Raising the Level of Abstraction in Architectural Simulation. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 307–318, Feb. 2010.

[81] B. Goel, S. McKee, R. Gioiosa, K. Singh, M. Bhadauria, and M. Cesati. Portable, scalable, per-core power estimation for intelligent resource management. In *International Green Computing Conference*, pages 135 –146, aug. 2010. doi: 10.1109/GREENCOMP.2010.5598313.

[82] R. L. Graham. Bounds of Certain Multiprocessing Anomalies. *The Bell System Technical Journal*, XLV:1563–1581, November 1966.

[83] R. L. Graham, T. S. Woodall, and J. M. Squyres. Open MPI: A Flexible High Performance MPI. In *Proceedings, 6th Annual International Conference on Parallel Processing and Applied Mathematics*, Poznan, Poland, September 2005.

[84] M. Grannæs. *Reducing Memory Latency by Improving Resource Utilization*. PhD thesis, Norwegian University of Science and Technology, Department of Computer and Information Science, 2010.

[85] I. Grasso, P. Radojkovic, N. Rajovic, I. Gelado, and A. Ramirez. Energy Efficient HPC on Embedded SoCs: Optimization Techniques for Mali GPU. In *IEEE 28th International Parallel and Distributed Processing Symposium*, pages 123–132, May 2014. doi: 10.1109/IPDPS.2014.24.

[86] T. Grosser, A. Groesslinger, and C. Lengauer. Polly Ů Performing Polyhedral Optimizations on a Low-Level Intermediate Representation. *Parallel Processing Letters*, 22(04), 2012. doi: 10.1142/S0129626412500107. URL http://www.worldscientific.com/doi/abs/10.1142/S0129626412500107.

[87] A. Gutierrez, R. Dreslinski, and T. Mudge. Evaluating private vs. shared last-level caches for energy efficiency in asymmetric multi-cores. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 191–198, July 2014. doi: 10.1109/SAMOS.2014.6893211.

[88] W. Haensch, E. Nowak, R. Dennard, P. Solomon, A. Bryant, O. Dokumaci, A. Kumar, X. Wang, J. Johnson, and M. Fischetti. Silicon CMOS devices beyond scaling.

IBM Journal of Research and Development, 50(4.5):339–361, July 2006. ISSN 0018-8646. doi: 10.1147/rd.504.0339.

[89] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program analysis. In *Journal of Instruction Level Parallelism*, 2005.

[90] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, Mar. 2000. ISSN 0272-1732. doi: 10.1109/40.848474. URL http://dx.doi.org/10.1109/40.848474.

[91] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011. ISBN 012383872X, 9780123838728.

[92] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPlib: The BSP Programming Library. *Parallel Computing*, 24(14):1947–1980, Dec. 1998. ISSN 0167-8191. doi: 10.1016/S0167-8191(98)00093-3. URL http://dx.doi.org/10.1016/S0167-8191(98)00093-3.

[93] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, July 2008. ISSN 0018-9162. doi: 10.1109/MC.2008.209. URL http://dx.doi.org/10.1109/MC.2008.209.

[94] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29: 1170–1183, December 1986. ISSN 0001-0782.

[95] M. Horowitz, T. Indermaur, and R. Gonzalez. Low-power digital design. In *IEEE Symposium on Low Power Electronics, 1994. Digest of Technical Papers.*, pages 8–11, Oct 1994. doi: 10.1109/LPE.1994.573184.

[96] A. C. Iordan and L. Natvig. Energy Efficient Methods for Multi-Core Programming. In *6th summer school on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems (ACACES 2010)*, July 2010.

[97] A. C. Iordan and L. Natvig. Task Based Programming on Multicores, A Case Study on Energy Consumption. In *Third Swedish Workshop on Multi-Core Computing MCC'10*, November 2010.

[98] A. C. Iordan, A. Podobas, L. Natvig, and M. Brorsson. Investigating the Potential of Energy-savings Using a Fine-grained Task Based Programming Model on Multicores. Presented at the 2nd Workshop on Applications for Multi and Many Core Processors (A4MMC 2011), June 2011.

[99] A. C. Iordan, M. Jahre, and L. Natvig. Towards Efficient Simulation of Task Based Parallel Applications. In *Proceedings of the Norsk Informatikkonferanse NIK 2012*, November 2012.

[100] A. C. Iordan, M. Jahre, and L. Natvig. Challenges of Reducing Cycle-accurate Simulation Time for TBP Applications. *Procedia Computer Science*,

18:1814 – 1823, 2013. ISSN 1877-0509. doi: http://dx.doi.org/10.1016/j.procs.2013.05.350. URL http://www.sciencedirect.com/science/article/pii/S1877050913004936. 2013 International Conference on Computational Science.

[101] A. C. Iordan, M. Jahre, and L. Natvig. On the Energy Footprint of Task Based Parallel Applications. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 164–171, July 2013. doi: 10.1109/HPCSim.2013.6641409.

[102] A. C. Iordan, M. Jahre, and L. Natvig. Victim Selection Policies for Intel TBB: Overheads and Energy Footprint. In E. Maehle, K. Römer, W. Karl, and E. Tovar, editors, *Architecture of Computing Systems - ARCS 2014*, volume 8350 of *Lecture Notes in Computer Science*, pages 13–24. Springer International Publishing, 2014. ISBN 978-3-319-04890-1. doi: 10.1007/978-3-319-04891-8_2. URL http://dx.doi.org/10.1007/978-3-319-04891-8_2.

[103] A. C. Iordan, M. Jahre, and L. Natvig. Tuning the victim selection policy of Intel TBB. *Journal of Systems Architecture*, 61(10): 584 – 591, 2015. ISSN 1383-7621. doi: http://dx.doi.org/10.1016/j.sysarc.2015.07.004. URL http://www.sciencedirect.com/science/article/pii/S1383762115000740.

[104] M. Jahre. *Managing Shared Resources in Chip Multiprocessor Memory Systems*. PhD thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2010.

[105] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 6:1–6:11, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3247-7. doi: 10.1145/2676870.2676883. URL http://doi.acm.org/10.1145/2676870.2676883.

[106] S. Kapil. UltraSPARC Gemini: Dual CPU Processor. http://www.hotchips.org/archive/, 2003. (retrieved November 2016).

[107] S. Kaxiras and M. Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan and Claypool Publishers, 2008.

[108] C. Kessler and J. Keller. Models for Parallel Computing: Review and Perspectives. In *PROCEEDINGS, PARS*, pages 13–29, 2007.

[109] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *IEEE Comput. Archit. Lett.*, 1(1), 2002.

[110] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded

Sparc processor. *Micro, IEEE*, 25(2):21–29, March 2005. ISSN 0272-1732. doi: 10.1109/MM.2005.35.

[111] M. Korch and T. Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms: Research articles. *Concurr. Comput. : Pract. Exper.*, 16 (1):1–47, 2004. ISSN 1532-0626. doi: http://dx.doi.org/10.1002/cpe.v16:1.

[112] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 81–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2043-X. URL http://dl.acm.org/citation.cfm?id=956417.956569.

[113] R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. Heterogeneous Chip Multiprocessors. *Computer*, 38(11):32–38, Nov 2005. ISSN 0018-9162. doi: 10.1109/MC.2005.379.

[114] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core Architecture Optimization for Heterogeneous Chip Multiprocessors. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 23–32, New York, NY, USA, 2006. ACM. ISBN 1-59593-264-X. doi: 10.1145/1152154.1152162. URL http://doi.acm.org/10.1145/1152154.1152162.

[115] P. Kurp. Green Computing. *Communications of the ACM*, 51(10):11–13, Oct. 2008. ISSN 0001-0782. doi: 10.1145/1400181.1400186. URL http://doi.acm.org/10.1145/1400181.1400186.

[116] G. Kyriazis. Heterogeneous System Architecture: A Technical Review. http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/hsa10.pdf. (retrieved November 2016).

[117] A. J. Lande. Evaluering av Chip Multiprosessor Simulatorer. http://www.diva-portal.org/smash/get/diva2:350431/FULLTEXT01.pdf, 2006. Norwegian University of Science and Technology, Department of Computer and Information Science.

[118] C. Lattner. LLVM and Clang: Advancing Compiler Technology, Presentation at Free and Open Source Developers' European Meeting (FOSDEM), Brussels, Belgium. http://llvm.org/pubs/2011-02-FOSDEM-LLVMAndClang.pdf, 2011. (retrieved November 2016).

[119] D. Leijen, W. Schulte, and S. Burckhardt. The Design of a Task Parallel Library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-

0. doi: 10.1145/1640089.1640106. URL http://doi.acm.org/10.1145/1640089.1640106.

[120] D. Leijen, W. Schulte, and S. Burckhardt. The Design of a Task Parallel Library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640106. URL http://doi.acm.org/10.1145/1640089.1640106.

[121] C. E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010. ISSN 0920-8542. doi: 10.1007/s11227-010-0405-3. URL http://dx.doi.org/10.1007/s11227-010-0405-3.

[122] J. Li and J. Martínez. Power-Performance Considerations of Parallel Computing on Chip Multiprocessors. *ACM Trans. Archit. Code Optim.*, 2, 2005.

[123] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multi-Core and Many-Core Architectures. In *International Symposium on Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM*, pages 469–480, Dec 2009.

[124] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: Using Low-power Processors in Smartphones Without Knowing Them. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 13–24, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2150979. URL http://doi.acm.org/10.1145/2150976.2150979.

[125] F. X. Lin, Z. Wang, and L. Zhong. K2: A Mobile Operating System for Heterogeneous Coherence Domains. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 285–300, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541975. URL http://doi.acm.org/10.1145/2541940.2541975.

[126] G. Liu, J. Park, and D. Marculescu. Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems. In *ICCD*, pages 54–61. IEEE Computer Society, 2013. ISBN 978-1-4799-2987-0. URL http://dblp.uni-trier.de/db/conf/iccd/iccd2013.html#LiuPM13.

[127] D. J. MacKay. *Information Theory, Inference and Learning Algorithms.Chapter 20: An Example Inference Task: Clustering*. Cambridge University Press, 2003.

[128] MacQueen, J.B. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297,

Berkeley, California, 1967. University of California Press. URL http://projecteuclid.org/euclid.bsmsp/1200512992.

[129] A. Maghazeh, U. D. Bordoloi, P. Eles, and Z. Peng. General Purpose Computing on Low-Power Embedded GPUs: Has it Come of Age? In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII)*, pages 1–10, July 2013. doi: 10.1109/SAMOS.2013.6621099.

[130] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero. Overlapping Communication and Computation by Using a Hybrid MPI/SMPSs Approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 5–16, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0018-6. doi: 10.1145/1810085.1810091. URL http://doi.acm.org/10.1145/1810085.1810091.

[131] A. Marowka. TBBench: A Micro-Benchmark Suite for Intel Threading Building Blocks. *JIPS*, 8(2), 2012.

[132] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Comput. Archit. News*, 33(4):92–99, Nov. 2005. ISSN 0163-5964. doi: 10.1145/1105734.1105747. URL http://doi.acm.org/10.1145/1105734.1105747.

[133] T. Maruyama. *SPARC64 VI: Fujitsu's Next Generation Processor*, 2003. Microprocessor Forum.

[134] W. F. McColl. BSP Programming. In G. E. Blelloch, K. M. Chandy, and S. Jagannathan, editors, *Specification of Parallel Algorithms. Proceedings of DIMACS Workshop*, volume 18 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 21–35. American Mathematical Society, 1994.

[135] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. Frameworks for multi-core architectures: a comprehensive evaluation using 2D/3D image registration. In *Proceedings of the 24th International Conference on Architecture of Computing Systems*, 2011.

[136] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, Jan 2010. doi: 10.1109/HPCA.2010.5416635.

[137] S. Mittal. A Survey of Techniques for Architecting and Managing Asymmetric Multicore Processors. *ACM Computing Surveys*, 48(3):45:1–45:38, Feb. 2016. ISSN 0360-0300. doi: 10.1145/2856125. URL http://doi.acm.org/10.1145/2856125.

[138] G. Moore. Cramming More Components Onto Integrated Circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998. ISSN 0018-9219. doi: 10.1109/JPROC.1998.658762.

[139] T. Y. Morad, U. C. Weiser, A. Kolodnyt, M. Valero, and E. Ayguade. Performance, Power Efficiency and Scalability of Asymmetric Cluster Chip Multiprocessors. *IEEE Computer Architecture Letters*, 5(1):14–17, Jan 2006. ISSN 1556-6056. doi: 10.1109/L-CA.2006.6.

[140] S. Mukherjee, Y. Sun, P. Blinzer, A. K. Ziabari, and D. Kaeli. A comprehensive performance analysis of HSA and OpenCL 2.0. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 183–193, April 2016. doi: 10.1109/ISPASS.2016.7482093.

[141] L. Natvig and A. C. Iordan. Green Computing: Saving Energy by Throttling, Simplicity and Parallelization. *UPGRADE : The European Journal for the Informatics Professional*, XII(4), 2011.

[142] C. J. Newburn, B. So, Z. Liu, M. McCool, A. Ghuloum, S. D. Toit, Z. G. Wang, Z. H. Du, Y. Chen, G. Wu, P. Guo, Z. Liu, and D. Zhang. Intel's Array Building Blocks: A Retargetable, Dynamic Compiler and Embedded Language. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 224–235, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-1-61284-356-8. URL http://dl.acm.org/citation.cfm?id=2190025.2190069.

[143] K. Olukotun and L. Hammond. The Future of Microprocessors. *Queue*, 3(7):26–29, Sept. 2005. ISSN 1542-7730. doi: 10.1145/1095408.1095418. URL http://doi.acm.org/10.1145/1095408.1095418.

[144] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-chip Multiprocessor. *SIGPLAN Not.*, 31(9):2–11, Sept. 1996. ISSN 0362-1340. doi: 10.1145/248209.237140. URL http://doi.acm.org/10.1145/248209.237140.

[145] H. Pan, B. Hindman, and K. Asanović. Composing Parallel Software Efficiently with Lithe. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2010.

[146] D. Patterson. The Trouble With Multicore. *IEEE Spectrum*, 47(7), 2010. ISSN 0018-9235.

[147] D. A. Patterson. Latency lags bandwith. *Commun. ACM*, 47(10):71–75, Oct. 2004. ISSN 0001-0782. doi: 10.1145/1022594.1022596. URL http://doi.acm.org/10.1145/1022594.1022596.

[148] E. Perelman, G. Hamerly, and B. Calder. Picking Statistically Valid and Early Simulation Points. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, pages 244–, Washington,

DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2021-9. URL http://dl.acm.org/citation.cfm?id=942806.943854.

[149] J. Perez, R. Badia, and J. Labarta. A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures. In *IEEE International Conference on Cluster Computing*, 2008.

[150] C. Pheatt. Intel Threading Building Blocks. *J. Comput. Sci. Coll.*, 23 (4):298–298, Apr. 2008. ISSN 1937-4771. URL http://dl.acm.org/citation.cfm?id=1352079.1352134.

[151] S. Pllana and F. Xhafa, editors. *Programming multi-core and many-core computing systems*, chapter Multi- and Many-cores, Architectural Overview for Programmers. Wiley, 2016. In print.

[152] A. Podobas. *Improving Performance and Quality-of-Service through the Task-Parallel Model: Optimizations and Future Directions for OpenMP*. PhD thesis, KTH, Software and Computer systems, SCS, September 2015.

[153] A. Podobas, M. Brorsson, and K.-F. Faxén. A Comparison of Some Recent Task-based Parallel Programming Models. In *Third Workshop on Programmability Issues for Multi-Core Computers*, 2009.

[154] A. Prakash, S. Wang, A. E. Irimiea, and T. Mitra. Energy-Efficient Execution of Data-Parallel Applications on Heterogeneous Mobile Platforms. In *IEEE International Conference on Computer Design (ICCD)*, pages 208–215, Oct 2015. doi: 10.1109/ICCD.2015.7357105.

[155] N. Rajovic, A. Rico, J. Vipond, I. Gelado, N. Puzovic, and A. Ramirez. Experiences with Mobile Processors for Energy Efficient HPC. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 464–468, San Jose, CA, USA, 2013. EDA Consortium. ISBN 978-1-4503-2153-2. URL http://dl.acm.org/citation.cfm?id=2485288.2485400.

[156] N. Rajovic, L. Vilanova, C. Villavieja, N. Puzovic, and A. Ramirez. The low power architecture approach towards exascale computing. *Journal of Computational Science*, 4(6):439 – 443, 2013. ISSN 1877-7503. doi: http://dx.doi.org/10.1016/j.jocs.2013.01.002. URL http://www.sciencedirect.com/science/article/pii/S1877750313000148.

[157] N. Rajovic, A. Rico, N. Puzovic, C. Adeniyi-Jones, and A. Ramirez. Tibidabo: Making the case for an ARM-based HPC system. *Future Generation Computer Systems*, 36:322 – 334, 2014. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2013.07.013. URL http://www.sciencedirect.com/science/article/pii/S0167739X13001581.

[158] A. Ramirez. The Mont-Blanc prototype. https://www.montblanc-project.eu/sites/default/files/publications/

20140522montblanc-prototypes-workshop.pdf. (retrieved November 2016).

[159] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramirez, and M. Valero. On the simulation of large-scale architectures using multiple application abstraction levels. *ACM Trans. Archit. Code Optim.*, 8(4), 2012.

[160] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. *SIGARCH Comput. Archit. News*, 37(3):371–382, June 2009. ISSN 0163-5964. doi: 10.1145/1555815.1555801. URL http://doi.acm.org/10.1145/1555815.1555801.

[161] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai. Challenges in Computer Architecture Evaluation. *Computer*, 36(8):30–36, Aug. 2003. ISSN 0018-9162. doi: 10.1109/MC.2003.1220579. URL http://dx.doi.org/10.1109/MC.2003.1220579.

[162] L. Spracklen and S. Abraham. Chip multithreading: Opportunities and Challenges. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 248–252, Feb 2005. doi: 10.1109/HPCA.2005.10.

[163] L. Stanisic, B. Videau, J. Cronsioe, A. Degomme, V. Marangozova-Martin, A. Legrand, and J.-F. Méhaut. Performance Analysis of HPC Applications on Low-power Embedded Platforms. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, pages 475–480, San Jose, CA, USA, 2013. EDA Consortium. ISBN 978-1-4503-2153-2. URL http://dl.acm.org/citation.cfm?id=2485288.2485403.

[164] B. J. Svensson and M. Sheeran. Parallel Programming in Haskell Almost for Free: An Embedding of Intel's Array Building Blocks. In *Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-performance Computing*, FHPC '12, pages 3–14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1577-7. doi: 10.1145/2364474.2364477. URL http://doi.acm.org/10.1145/2364474.2364477.

[165] TBB. Intel Threading Building Blocks. http://software.intel.com/sites/products/documentation/hpc/tbb/getting_started.pdf, . (retrieved November 2016).

[166] TBB. Intel Corporation. *Intel Threading Building Blocks Reference Manual*. Available: http://threadingbuildingblocks.org/, . (retrieved November 2016).

[167] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25, Jan. 2002. ISSN 0018-8646. doi: 10.1147/rd.461.0005. URL http://dx.doi.org/10.1147/rd.461.0005.

[168] M. Tremblay, J. Chan, S. Chaudhry, A. Conigliam, and S. Tse. The MAJC architecture: a synthesis of parallelism and scalability. *Micro, IEEE*, 20(6):12–25, Nov 2000. ISSN 0272-1732. doi: 10.1109/40.888700.

[169] N. Vachharajani, M. Iyer, C. Ashok, M. Vachharajani, D. I. August, and D. Connors. Chip Multi-processor Scalability for Single-threaded Applications. *SIGARCH Comput. Archit. News*, 33(4):44–53, Nov. 2005. ISSN 0163-5964. doi: 10.1145/1105734.1105741. URL http://doi.acm.org/10.1145/1105734.1105741.

[170] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, Aug. 1990. ISSN 0001-0782. doi: 10.1145/79173.79181. URL http://doi.acm.org/10.1145/79173.79181.

[171] L. G. Valiant. A Bridging Model for Multi-Core Computing. *Journal of Computer and System Sciences*, 77(1):154 – 166, 2011. ISSN 0022-0000. doi: http://dx.doi.org/10.1016/j.jcss.2010.06.012. URL http://www.sciencedirect.com/science/article/pii/S0022000010000966.

[172] H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos. Parallel Programming of General-Purpose Programs Using Task-Based Programming Models. In *Proc. of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, 2011.

[173] A. L. Varbanescu. *On the Effective Parallel Programming of Multi-Core Processors*. PhD thesis, TU Delft, December 2010.

[174] A. Venkat and D. M. Tullsen. Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor. In *ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 121–132, June 2014. doi: 10.1109/ISCA.2014.6853218.

[175] D. B. Wagner and B. G. Calder. Leapfrogging: a portable technique for implementing efficient futures. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 208–217, New York, NY, USA, 1993. ACM. ISBN 0-89791-589-5. doi: http://doi.acm.org/10.1145/155332.155354. URL http://doi.acm.org/10.1145/155332.155354.

[176] D. B. Wagner and B. G. Calder. Leapfrogging: a portable technique for implementing efficient futures. In *Proc. of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1993.

[177] D. W. Wall. Limits of Instruction-level Parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 176–188, New York, NY, USA, 1991. ACM. ISBN 0-89791-380-9. doi: 10.1145/106972.106991. URL http://doi.acm.org/10.1145/106972.106991.

[178] A. Weissel and F. Bellosa. Process cruise control: event-driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '02, pages 238–246, New York, NY, USA, 2002. ACM. ISBN 1-58113-575-0.

[179] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. Simulation sampling with live-points. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 2–12, March 2006. doi: 10.1109/ISPASS.2006.1620785.

[180] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro*, 26(4):18–31, July 2006. ISSN 0272-1732. doi: 10.1109/MM.2006.79.

[181] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, Mar. 1995. ISSN 0163-5964. doi: 10.1145/216585.216588. URL http://doi.acm.org/10.1145/216585.216588.

[182] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Statistical Sampling of Microarchitecture Simulation. *ACM Trans. Model. Comput. Simul.*, 16(3):197–224, July 2006. ISSN 1049-3301. doi: 10.1145/1147224.1147225. URL http://doi.acm.org/10.1145/1147224.1147225.

[183] S. L. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks. Quantifying sources of error in McPAT and potential impacts on architectural studies. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 577–589, Feb 2015. doi: 10.1109/HPCA.2015.7056064.

[184] J. J. Yi and D. J. Lilja. Simulation of Computer Architectures: Simulators, Benchmarks, Methodologies, and Recommendations. *IEEE Trans. Comput.*, 55(3), 2006.

[185] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and Comparing Prevailing Simulation Techniques. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 266–277, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2275-0. doi: 10.1109/HPCA.2005.8. URL http://dx.doi.org/10.1109/HPCA.2005.8.

Appendix A

Paper A.3

Investigating the Potential of Energy-savings Using a Fine-grained Task Based Programming Model on Multi-cores

Alexandru Iordan, Artur Podobas, Lasse Natvig and Mats Brorsson

2nd Workshop on Applications for Multi and Many Core Processors (A4MMC)
2011

Abstract

In this paper we study the relation between energy-efficiency and parallel executions when implemented with a fine-grained task-centric programming model. Using a simulation framework comprised of an architectural simulator and a power and area estimation tool, we have investigated the potential energy-savings when employing parallelism on multi-cores system. In our experiments with 2 - 8 multi-cores systems, we employed frequency and voltage scaling in order to keep the relative performance of the systems constant and measured the energy-efficiency using the *Energy-delay-product*. Also, we compared the energy consumption of the parallel execution against the serial one. Our results show that through judicious choice of load balancing parameters, significant improvements of around 200 % in energy consumption can be acheived.

A.1 Introduction

At the start of the new millennium, with performance being limited by high power budgets and heat dissipations requirements, the superscalar paradigm reached the point of diminishing returns. Faced with the constraint of *the power wall*[12], hardware developers were in need of new ways to efficiently use the ever increasing transistor count predicted by Moore's law.

Multi-core architectures have been a natural solution for the power wall: several less complex and significantly less "power hungry" cores are integrated on a single processor. Processors like Sun's Niagara T1 and T2, Tilera's Tile64 or IBM's Cyclops-64 use less complex cores, with shallow pipelines and simpler branch prediction, and lower clock speeds than previous CPU generations. Scaling down frequency (f) and supply voltage (V_{dd}) has a large effect on the chip's dynamic power, as shown by the relation: $power_{dynamic} \sim V_{dd}^2 \cdot f$. Developing parallel applications, that can take advantage of such chips has the potential of reducing energy consumption while still providing high performance.

This paper is a initial study of the impact of load-balancing on energy-efficiency in parallel executions. We use a very simple test scenario in which we increase the number of cores while proportionally decreasing their working speed. The applications in our experiments are developed using a paradigm called *Task Based Programming* (TBP). TBP organizes an application as a set of computational units (called *tasks*) that are scheduled across different cores. Parallel applications developed with TBP are known to handle irregular dependencies on input sets well and can adapt to varying computational load[16]. The base concept of the TBP model is that the programmer should identify and annotate pieces of code (tasks) which can be executed concurrently with other tasks, while the complexity of the hardware is abstracted away from him/her.

Generally, parallelization of applications has a cost of added overhead that sometimes scales badly, and this can increase the amount of energy that is used. In our study we employed a rather simple core design which we did not modify as we scaled up the number of cores. Our goal was not to find a way to develop a multi-core that was highly energy-efficient, but to investigate the potential to save energy by parallel executions. We used voltage and frequency scaling to keep the relative performance of the systems constant and to maintain the chip's power requirements under realistic values.

The rest of this paper is organized as follows: in Section 2 we give a brief introduction to power metrics, TBP and Wool library. Section 3 outlines the experimental methodology used in our experiments and Section 4 discusses our results. Section 5 presents related work and Section 6 describes our plans for future work. Section 7 concludes the paper.

A.2 Background

A.2.1 Power metrics

A well known metric that can balance performance and power requirement is *Performance^N/Watt*. The *N* parameter is used to increase or decrease the importance of the *Performance* component of the metric. For *N = 1*, this metric is used in the Green 500 list to rank world's most energy-efficient supercomputers[1].

Industry standard benchmarks like EnergyBench (for embedded systems) and SPECpower (for servers and multi-processor computers) use customized metrics to report on energy-efficiency. For example, SPECpower ranks a system using ssj_ops/Watt metric (stands for server side Java operations performed per Watt). This is a derived form of the *Performance/Watt* metric and it represents the number of the executed SPEC operations divided by the average power of the system.

Another frequently used metric is the *Energy-delay-product* (EDP). When comparing scenarios that do not alter instruction count, this metric is equivalent to the reciprocal of *Performance^2/Watt*. Offering equal weight to energy consumption and performance, EDP ensures a balanced energy-efficiency comparison among the test systems. Since our focus is on studying the trade-off of energy-savings and performance on multi-core systems, we choose to use EDP for our experiments.

A.2.2 Task Based Programming

A *task*, which can be fine-grained or coarse-grained, is a section of the code that performs some operations over a set of parameters[15]. *Task Based Programming* (TBP) is a programming paradigm that allows the parallelization of applications that can be divided in multiple tasks. TBP comes in contrast with *data parallelism* where the same operations are executed by different nodes (or cores) with different data[14]. Using TBP model, the programmer should identify pieces of code (tasks) to be executed concurrently with other tasks.

TBP libraries, like Intel's TBB[21] and Cilk++[7] increases the productivity of programmers since details like distribution of work to a set of cores and message passing between parallel processes are abstracted away from the programmer. etails of the system. Membarth et al.[19] performed a comparative study of several frameworks for parallel programming on multi-cores. Their results showed that TBP libraries like Intel's TBB and Cilk++ not only perform better than other frameworks like OpenMP or OpenCL, but also have a wide usability and provide a better productivity for the programmer.

A.2.3 The Wool library

Our approach involves a lightweight TBP library called Wool [11]. Wool is a work-stealing parallel library that was designed with the ability to scale well with small tasks (smaller than a hundred cycles). These characteristics make Wool very efficient in dealing with work imbalance and also assure that it has very low overheads. The comparative study in [20] shows that Wool outperforms some other parallelization libraries (like Cilk++ or OpenMP) in terms of cycle costs for parallelization and task management operations.

Wool uses special data structures called *task queues* to store, manage and schedule the tasks for each worker thread. This differs from other models such as the GCC version of OpenMP which have one global queue containing all the tasks. Private queues generally improve performance of the system, since the locking-contention usually is much smaller compared to global queues. However, distributed queues face the challenge of balancing the work among them. More details and examples about Wool can be found in [11] and [20].

In Wool, load-balancing is implemented through the *randomly task stealing* technique. When a worker thread empties its own task queue, the stealing mechanism sweeps through all available worker queues to find available tasks to steal. Because making a task *stealable* adds an overhead, the programmer can control the number of stealable tasks per queue. In this way, the programmer has control over the load-balancing of the application.

Another scenario when task stealing is employed is when a worker is trying to synchronize with one of its children, and finds it stolen. In order to prevent threads from being idle a technique called *leap-frogging* is used. More details about this technique can be found in [22].

A.3 Experimental setup

A.3.1 Architectural simulations

Using the M5 simulator [5], we performed full-system simulations of several multi-core platforms. In a full-system simulation, the target system is able to run its own operating system and in our experiments we used the 2.6.27 Linux kernel. This type of simulation also makes it possible to record the behavior of all key components of the system: core, cache hierarchy, memory controller, main memory.

The basis for our modeled CPU is the Alpha 21364 processor from DEC. The choosing of this model was motivated by the fact that Alpha ISA is the most stable one for full-system multi-core simulations in M5 [4]. A second reason is that the 21364 was also validated in McPAT [18] (more details in section 3.2). In all multi-core systems simulated we used

the same processor architecture with all parameters kept constant, except the three main parameters: number of cores, supply voltage and core frequency. We assumed voltage and frequency scaling at chip level so that we keep the power requirement under realistic values. We used the formula *core_frequency = single_core_frequency / number_of_cores* to maintain constant the relative performance of the test systems.

The maximum number of cores we simulated was limited only by the values for core frequency and V_{dd} (the way we calculated our voltage scaling values is described in section 3.2). The minimum value for V_{dd} we could assign was $2.3 * V_{th}$ [17] and in our case this limit is $0.19 * 2.3 = 0.44$ V.

The original 21364 has a clock frequency of 1.2 GHz and was produced in 180 nm technology. We assumed a 65 nm process technology and by linear scaling, similarly to Li and Martinez [17], we can determined the single-core frequency at 3.32 GHz. However, using this frequency for the single-core system and then scaling down V_{dd} results into voltage values very close to the $2.3 * V_{th}$ limit. For such low values of V_{dd} the chip's leakage current increases significantly and the static power can become dominant. To alleviate this we chose to assign a lower core frequency of 2 GHz to the single-core system. Since the speed of the cores does not affect the way they process the applications, just the execution time, the trends presented in our results are the same for both sets of experiments (the ones with a single-core running at 3.32 GHz and the ones with the single-core running at 2 GHz).

Table A.1 lists the main characteristics of our experiments. Further details about M5's architectural parameters and characteristics listed in Table A.1 and Table A.2 can be found in [4] and [5].

The modeled system uses a cache hierarchy with split data and instruction private L1 caches. All cores share a 2 MB on-chip L2 cache through a common bus and implement a MOESI cache coherence protocol. Details about the cache hierarchy are given in Table A.2.

A.3.2 Power estimations

Our simulation framework includes a power and area estimation tool called McPAT[18]. Developed in collaboration by HP-labs and the University of Notre Dame, McPAT models all major system components of a computer system (including in-order and out-of-order cores, network-on-chip, shared and private caches, memory controllers). Using information from the ITRS 2007 roadmap [2], McPAT supports design space exploration for single and multi-core architectures ranging from 90 nm to 22 nm production technology.

According to the *Process Integration, Devices, and Structures* chapter in ITRS 2007, there are 3 types of circuit logic: high performance (HP), low operating power (LOP) and low standby power (LSTP). HP devices include chips of high complexity and performance such as the microprocessors for desktop or server computers. LOP devices include

Table A.1 Main characteristics of modeled processor core

Parameter	Value
Process technology	65 nm
Nominal V_{dd}	1.1 / 0.89 / 0.82 / 0.78 / 0.76 / 0.74 / 0.73 / 0.72 V
V_{th}	0.19 V
Type of execution	Out-of-Order
Instruction set	Alpha
No. Cores	1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 cores
Clock frequency	2000 / 1000 / 666 / 500 / 400 / 333 / 285 / 250 MHz
Fetch/issue/commit width	4 / 4 / 4 insts./cycle
Inst. window size (Int / FP)	20 / 15 entries
Functional units	4 integer ALUs 1 integer multiply / divide 1 FP ALU 1 FP multiply/divide

Table A.2 Cache parameters

Cache	Size	Assoc. (bits)	Block size (bits)	Access latency	MSHRs Targets / MSHR	Banks
L1 private iCache	32 KB	2	64	2	4 MSHRs /4 tgts	1
L1 private dCache	32 KB	4	64	2	4 MSHRs /4 tgts	1
L2 shared Cache	2 MB	4	64	10	4 MSHRs /4 tgts	4

relatively high-performance mobile circuits, like those in notebooks. LSTP devices are typically intended for low performance applications like cellular phones. We performed our estimations for the 65 nm technology and the HP device type.

McPAT is able to report both dynamic and static power. Dynamic power for each system component is defined as: $power_{dynamic} \sim AF \cdot C \cdot V_{dd}^2 \cdot f$, where *AF* is the activity factor, *C* is the total load capacitance, V_{dd} is the supply voltage and f is the clock frequency. *AF* is estimated using access statistics and component's characteristics provided by the architectural simulation of that component. The capacitance is computed with analytic models for each basic circuit block that makes up the system component. In addition to the dynamic power component McPAT also estimates the leakage power. As described in [18] the leakage current is estimated using MASTAR[3] and data from Intel.

The developers of McPAT validated the Alpha 21364 processor model we used in our simulations against published data. As the results in [18] show, McPAT is able to estimate the power requirements of the Alpha 21364 with an average error of 21 % (this value is highly influnced by the fact that validation was done for peak power and not average power).

In order to correlate the values for frequency and voltage scaling we resorted to a report from Intel[9] documenting this relation (in steps of 200 MHz) for three Pentium proces-

sors. From that report we extracted an average voltage step of 0.052 V per 200 MHz. This represents 3.88 % of the 1.34 V nominal voltage used for that family of processors. Since we simulated CPUs in the 65 nm technology process which has a 1.1 V nominal voltage [2], we calculated the voltage scaling step as 3.88 % of this nominal voltage for a frequency step of 200 MHz. By subtracting each core frequency in Table A.1 from the single-core value and dividing the result by 200, we calculated the number of frequency steps. We used this number of steps to proportionally reduce V_{dd}. The resulting voltage values for each multi-core configuration we simulated are reported in Table A.1.

A.3.3 Benchmarks

In our experiments we used a subset of the Barcelona OpenMP Task benchmark Suite (BOTS)[10]. BOTS is a benchmark compilation assembled by the Barcelona Supercomputing Center (BSC) to assess the performance of task-based programming models. Some of the kernels come from other benchmark collections (like *FFT* or *strassen*) and the others were written by the team at BSC (like *sparselu*). For our experiments, we changed the default OpenMP parallelization to a Wool implementation. This change was motivated by our wish to use a lightweight library with a low parallelization overhead. To allow for reasonable simulation times (from 12 hours to 2 days), the workloads we used are generally smaller than real-life problem sizes. Table C.1 lists the size of the workloads we used in our experiments. The benchmarks have been cross-compiled for the Alpha ISA using a cross-compiler (consisting of gcc-4.3.2 and glibc-2.6.1) [4]. All benchmarks have been compiled using the flags: -O3 -static -pthread. A brief description of the BOTS subset that we used is given in section 4.

Table A.3 Input workloads used in experiments

	Alignment	FFT	Fib	nQueens	SparseLU	Strassen
Input	20 proteins	512x512 matrix of floats	40'th element	13x13 board	100x100 sparse matrix of 20x20 blocks	1024x1024 matrix

A.4 Results

The main two issues that need to be addressed in order to improve a system's energy-efficiency when running parallel applications are parallel overhead and load imbalance. In order to quantify the effect of the two causes on energy consumption, we performed experiments in which we altered Wool's ability to deal with load imbalance. By controlling the number of tasks that each worker queue is allowed to mark as stealable, we also controlled the task distribution across worker threads. Taking into consideration that an imbalanced task-tree will also force the worker threads to perform more "management

operations" (search for task to steal, successful/unsuccessful steals), modifying the number of stealable tasks also has effect on the overhead. We covered a wide range of testing points, from a minimum of 1 stealable task to 10000. There were some benchmarks (*alignment* and *sparselu*) for which the load imbalance for low values of stealable tasks was so large that its execution required an unreasonable long simulation time. For these applications we reduced the test range. For reasons of space limitations for this paper, Table A.4 only lists the lowest EDP values and the corresponding number of stealable tasks. For the same reason, we do not report or discuss performance-orientated metrics like speed-ups.

Alignment is a protein alignment benchmark that is based on the Myers and Miller algorithm. In a *master-slave* manner, all the tasks in its execution are spawned by the main worker thread (the thread that is used to start the program). The children tasks, which do not spawn any other tasks, need to be available for the other workers. That is why the number of stealable task has a big effect on the energy-efficiency of the system executing this benchmark. All the other threads need to steal work from the main thread and a low number of stealable task leads to race contention among them. Our experiments showed a progressive decrease in EDP as the number of stealable tasks increased up to a threshold of around 200 tasks. After this point the improvements come at a much slower pace. For the stealability parameters listed in Table A.4, the workload is almost perfectly balanced, and the energy consumption for all multi-core executions is below the single-core one (see Fig.A.1)

Fibonacci is a recursive benchmark that calculates the Fibonacci series of a given value n. The workload in the tasks are fine-grained, with leaf-nodes having only a single *if* and *return* statement. Using a recursive, divide-and-conquer approach, this application creates a very extensive task-tree which means that there is enough work for every worker. This application has good improvements when parallelized, with the workload evenly balanced among the workers even for low values of stealable tasks.

FFT calculates the Discrete Fourier Transform of a matrix in a recursive manner using the Cooley-Turkey algorithm. It showed good results when parallelized and all multi-core executions had an energy consumption under the single-core one (see Fig.A.1).

nQueens is a search-and-prune benchmark that generates all solution for the nQueens problem. It "builds" the solutions row by row and each valid position of a queen spawns a new task. Like Fibonacci, this is another application that benefits little from large numbers of stealable tasks. However nQueens does not generate a very large task-tree. The main worker thread starts the work by spawning a task for each valid position of a queen on the first row of the chessboard. The other workers steal these tasks and begin working with them. Since these tasks are coarse-grained, the workers need to steal fewer in order to keep busy. This application did not show improvements in energy consumption for all multi-core systems, as will be discussed later in the section.

SparseLU calculates the LU matrix factorization and the algorithm is fairly unbalanced. Just like Alignment, SparseLU spawns a relative small number of tasks and the challenge

is to schedule them in a balanced manner among the worker threads. With the right number of stealable tasks (see Table A.4), the workload imbalance is solved and the multi-core executions register a lower energy consumption than the single-core one.

Strassen is a parallel matrix multiplication algorithm. The algorithm subdivides the array into smaller arrays, and performs matrix multiplication on them. Like nQueens, Strassen showed partial improvements on energy-savings when parallelized. The parallel matrix multiplication executed faster on 2 and 3 cores compared to the 1 core system, but lost this advantage as the core count increased. Again, the reasons for this will be discussed later in the section.

Using the values for number of stealable tasks listed in Table A.4, we performed a comparison of the parallel execution against the serial one. With these values, the performance of each benchmark on the multi-core systems is at a maximum (at least from the load-balancing point of view), so we only measured *Energy* for this study. Fig.A.1 presents this comparison. As you can see, for most configurations, the parallel executions show a higher energy-efficiency (marked by a lower energy) than the serial one. The biggest improvement recorded is for *Fibonacci* which shows a 239 % decrease in energy consumption when comparing the 3-cores execution to the single-core one. However, after a certain point the descendant trend of energy consumption stops for all benchmarks. There are three reasons for this behavior.

First, as the number of cores grows — so too are the scheduling and task management overheads. We recorded maximum increases of 52 % (for *Strassen*) in instruction count going from 2 cores to 8 cores.

Second is the less than linear algorithmic speedup of the parallel programs. In addition to this, the work stealing mechanism can induce stalls and serialization into execution. There are situations when a worker thread is forced to wait the completion of a task that was stolen from him. The *leap-frogging* technique can alleviate this problem only if the stolen task spawns children. A simple quantification of these behaviors can be made by examining the increase in execution time: 20 % increase when going from 2-cores to 8-cores for *Strassen*.

Third reason is the increase in static power as we scale down V_{dd}. Frequency and voltage scaling have a positive impact on dynamic power (when increasing the core count from 2 cores to 8, all executions showed an average of 43 % decrease of dynamic power), but they have the opposit effect on static power (leakage power to be more precise). Leakage power is influenced mainly by fabrication parameters of the transistors (gate thickness, gate material etc.) but also by V_{dd} and V_{th} (the voltage at which the transistor is viewed as switched "on"). As we scale down V_{dd} and we get closer to the V_{th}, the leakage current increases and so is the static power. All multi-core configurations recorded an increase of 135 % of static power, when going from 2-cores to 8-cores.

Table A.4 Best EDP values for multi-core test systems

		2 cores	3 cores	4 cores	5 cores	6 cores	7 cores	8 cores
Alignment	EDP	40.76	34.98	38.57	45.99	49.89	59.41	62.19
	No. of steals	900	900	1000	900	1000	900	900
Fibonacci	EDP	218.5	161.56	177.29	199.53	201.73	211.81	224.92
	No. of steals	3	4	4	4	4	3	4
FFT	EDP	217.36	181.28	172.94	205.42	222.66	245.46	267.35
	No. of steals	900	1500	3000	3500	7000	10000	10000
nQueens	EDP	1453.35	1498.70	1708.82	1978.66	2239.21	2581.41	2829.59
	No. of steals	4	5	5	6	9	9	7
SparseLU	EDP	79.39	53.83	58.06	64.09	71.58	80.46	88.50
	No. of steals	200	200	900	900	900	900	1000
Strassen	EDP	37.85	37.79	37.45	46.52	56.11	69.11	83.32
	No. of steals	9	100	90	90	900	900	900

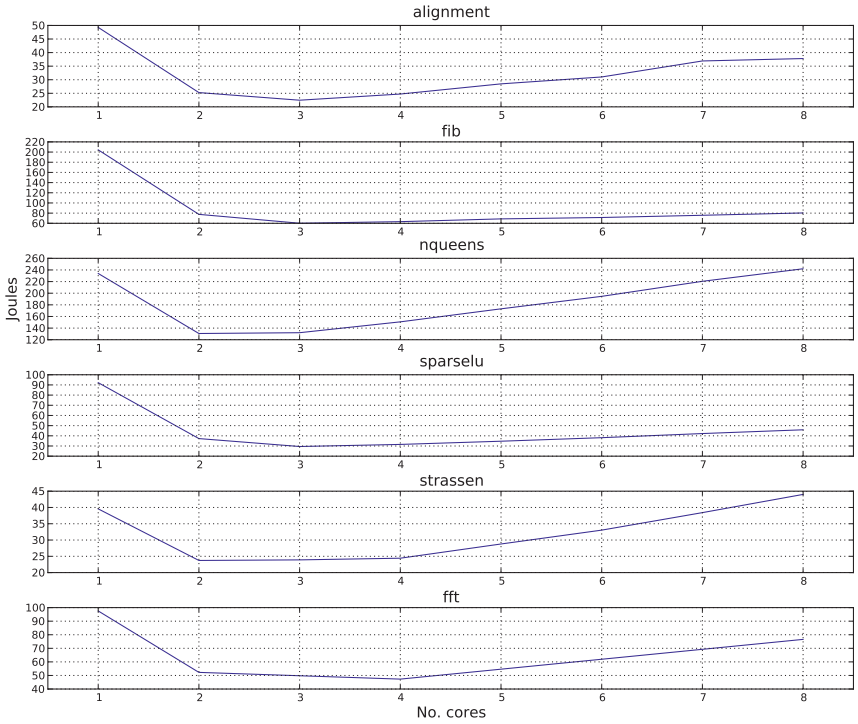


Figure A.1: Energy consumption comparison

A.5 Related work

There is a large body of work that focus on using parallel execution to improve performance and not energy consumption. What has got little interest thus far, at least to our knowledge, is quantifying the effect of TBP parallelization on energy efficiency on multi-core systems.

Li and Martinez[17] make a detailed power-performance exploration of parallel applications running on chip-multiprocessors. Using an analytical model, they perform extensive design space explorations to find the best multi-core configuration and also run simulations to verify the validity of the analytical results. They conclude that through judicious choice of parallelism's granularity and voltage/frequency scaling values, parallel computing can improve performance while maintaining or even reducing the power budget.

Contreras and Martonosi[8] make a study of Intel's *Threading Building Blocks* (TBB) and try to characterize some of the overheads associated with it. They emphasize the fact that this framework helps the programmer by abstracting away the complexity of the hardware. They also propose an improvement to TBB's task stealing mechanism in order to limit the parallelization overhead. Even though the focus of the authors is on performance, their implementation can also have a beneficial effect on the energy-efficiency of the system.

Sangyeun and Melhem[6] use an analytic framework to study the interplay between parallelism of an application, its performance and energy consumption. Their result demonstrate the advantage (quantified in energy or EDP) that can be gained from employing dynamic voltage and frequency scaling to execute the serial and parallel part of an application at different levels of frequency and voltage. Also they study the scenario when individual processors can be turned off when not in use. Even if it assumes an simplified environment, this work provides valuable theoretical insights into energy-aware resource management.

There is a number of papers that is very relevant for the future development of our study. However, since they are not directly related to the current stage of the research, they are referenced in the next section.

A.6 Future work

We want to extend our experiments towards a larger number of cores and at the same time change the processor model. We are currently investigating an ARM model which is a much recent and scalable multi-core architecture. Also, a 4-core ARM Cortex-A9 evaluation board is commercially available and that makes it possible to validate our model and also to do experiments with both simulation and real execution.

We also plan to do a more detailed characterization of the specific mechanisms employed by Wool. Future work will focus on a detailed quantification in terms of EDP of Wool's

load-balancing technique and possibly a comparison with other techniques (basic waiting, parking, etc. [11]).

Another approach to study the relation between task based parallelization and energy consumption is to use performance counters to track the behavior of real multi-cores. Weissel and Bellosa[23] and Goel et.al. [13] have been exploring power modeling with the use of performance counters. It is a long term goal for us to use similar approaches for achieving increased understanding of the energy issue, our models and their accuracy.

A.7 Conclusions

Although our study assumes a simplified environment and a simple test scenario we think it provides valid insights into how energy-efficiency and parallel execution relate. By integrating an architecture simulator (M5) with a power and area estimation tool (McPAT), we have put in place a framework for performance/power experiments. Using this framework, we studied the energy-efficiency of multi-core platforms running several BOTS benchmarks parallelized with the Wool library. Our experiments show improvements of the *EDP* metric when the parallel workload is balanced correctly for each benchmark and configuration.

Our experiments also show the potential for energy-efficiency improvements of parallel executions on multi-cores compared to the serial version of the same application on a single-core system. However, these improvements do not come for free. Task synchronization and management overhead, sub-linear speedups and increase in leakage power become more and more significant as the number of cores grows.

In all, we think the results we have found so far are promising and motivates for further research into energy-efficiency through parallelization.

Bibliography

[1] The Green 500 list. `http://www.green500.org/`. (retrieved November 2016).

[2] International Technology Roadmap for Semiconductors 2007 Edition. `http://www.itrs.net/links/2007itrs/ExecSum2007.pdf`. (retrieved November 2016).

[3] International Technology Roadmap for Semiconductors 2007 Edition, The Model for Assessment of CMOS Technologies and Roadmaps (MASTAR). `http://www.itrs.net/models.html`. (retrieved November 2016).

[4] The M5 Simulator System webpage. `http://www.m5sim.org/wiki/index.php/Main_Page`. (retrieved November 2016).

[5] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The M5 Simulator: Modeling Networked Systems. 26(4):52–60, July 2006. ISSN 0272-1732. doi: 10.1109/MM.2006.82. URL http://dx.doi.org/10.1109/MM.2006.82.

[6] S. Cho and R. G. Melhem. On the interplay of parallelization, program performance, and energy consumption. *IEEE Transactions on Parallel and Distributed Systems*, 21(3):342 –353, 2010.

[7] Cilk++. Cilk++: A quick, easy and reliable way to improve threaded performance. Available: http://software.intel.com/en-us/articles/intel-cilk-plus/. (retrieved November 2016).

[8] G. Contreras and M. Martonosi. Characterizing and Improving the Performance of Intel Threading Building Blocks. In *IEEE International Symposium on Workload Characterization*, pages 57–66, Sept 2008. doi: 10.1109/IISWC.2008.4636091.

[9] I. datasheet. Intel Pemtium M processor on 90 nm process with 2-MB L2 cache. http://download.intel.com/support/processors/mobile/pm/sb/30218908.pdf. (retrieved November 2016).

[10] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing*, pages 124–131, Sept 2009. doi: 10.1109/ICPP.2009.64.

[11] K.-F. Faxén. Wool - A Work Stealing Library. *SIGARCH Comput. Archit. News*, 36(5):93–100, June 2009. ISSN 0163-5964. doi: 10.1145/1556444.1556457. URL http://doi.acm.org/10.1145/1556444.1556457.

[12] S. Fuller and L. Millett. Computing Performance: Game Over or Next Level? *Computer*, 44(1):31–38, Jan 2011. ISSN 0018-9162. doi: 10.1109/MC.2011.15.

[13] B. Goel, S. McKee, R. Gioiosa, K. Singh, M. Bhadauria, and M. Cesati. Portable, scalable, per-core power estimation for intelligent resource management. In *International Green Computing Conference*, pages 135 –146, aug. 2010. doi: 10.1109/GREENCOMP.2010.5598313.

[14] W. D. Hillis and G. L. Steele, Jr. Data parallel algorithms. *Commun. ACM*, 29: 1170–1183, December 1986. ISSN 0001-0782.

[15] C. Kessler and J. Keller. Models for Parallel Computing: Review and Perspectives. In *PROCEEDINGS, PARS*, pages 13–29, 2007.

[16] M. Korch and T. Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms: Research articles. *Concurr. Comput. : Pract. Exper.*, 16(1): 1–47, 2004. ISSN 1532-0626. doi: http://dx.doi.org/10.1002/cpe.v16:1.

[17] J. Li and J. Martínez. Power-Performance Considerations of Parallel Computing on Chip Multiprocessors. *ACM Trans. Archit. Code Optim.*, 2, 2005.

[18] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multi-Core and Many-Core Architectures. In *International Symposium on Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM*, pages 469–480, Dec 2009.

[19] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. Frameworks for multi-core architectures: a comprehensive evaluation using 2D/3D image registration. In *Proceedings of the 24th International Conference on Architecture of Computing Systems*, 2011.

[20] A. Podobas, M. Brorsson, and K.-F. Faxén. A Comparison of Some Recent Task-based Parallel Programming Models. In *Third Workshop on Programmability Issues for Multi-Core Computers*, 2009.

[21] TBB. Intel Threading Building Blocks. `http://software.intel.com/sites/products/documentation/hpc/tbb/getting_started.pdf`. (retrieved November 2016).

[22] D. B. Wagner and B. G. Calder. Leapfrogging: a portable technique for implementing efficient futures. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 208–217, New York, NY, USA, 1993. ACM. ISBN 0-89791-589-5. doi: http://doi.acm.org/10.1145/155332.155354. URL `http://doi.acm.org/10.1145/155332.155354`.

[23] A. Weissel and F. Bellosa. Process cruise control: event-driven clock scaling for dynamic power management. In *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '02, pages 238–246, New York, NY, USA, 2002. ACM. ISBN 1-58113-575-0.

Appendix B

Paper B.1

Towards Efficient Simulation of Task Based Parallel Applications

Alexandru C. Iordan, Magnus Jahre and Lasse Natvig

Norsk Informatikkonferanse (NIK)
2012

Abstract

For computer architects and software developers, simulation is an indispensable tool for evaluating new designs and ideas. Unfortunately, it is significantly slower to simulate a parallel environment than to work on real hardware. In this work, we take a first step towards efficient simulation of Task Based Parallel (TBP) applications. Our key idea is that the number of completed tasks can be used as a work-related progress metric to sample these applications. Using this metric, we show that the complete execution of TBP programs can be accurately represented by simulating only a few samples. In fact, our TBP applications only need 5 samples on average to get a mean error below 5 %. In our experiments, when sampling at 1000 completed tasks, 5 samples correspond to 0.5 % of the total execution on average.

B.1 Introduction

Chip Multiprocessors (CMPs) or multi-core architectures [16] are becoming the prevalent computing infrastructure. Multi-cores were originally introduced as a means of avoiding the power wall. In particular, they make it possible to utilize the resources provided by improvements in production technology to increase aggregate performance without increasing the power budget [8]. Unfortunately for the programmer, this means that he can no longer expect to reap the benefits of Moore's law without parallelizing the application [17]. For this reason, there is a significant interest in programming models that simplify the task of parallelizing applications. In this work, we focus on *Task Based Programming (TBP)* which is a parallel programming model that has received significant attention recently [4, 7, 11, 22].

The key idea behind Task Based Programming (TBP) is that the programmer partitions the application into tasks and specifies the dependencies between them [7]. In some cases, the programming environment lets the programmer annotate procedure declarations in a way that makes the runtime system able to infer the task dependencies automatically [6, 19]. A task is a light weight unit of work, and all independent tasks can be executed in parallel. When a TBP application is run, a task management library distributes tasks to processing elements and ensures that all dependencies are met. Membarth et al. [14] found that TBP methodologies are among the most usable parallel programming models and that they yield good performance. In addition, the TBP model has a solid theoretical foundation. For instance, any greedy task scheduler will be within a factor of two of the optimal execution time [9].

Parallel programming is becoming a necessity for realizing the performance potential of processors, and new computer architectures should be evaluated with this in mind. On a high level, there are three main ways to evaluate new architectural ideas: analytical modeling, simulation and measurements on real hardware [21]. Analytical modeling has the advantage that it can efficiently cover large design spaces, but it can also be hard to reason about the accuracy of the findings. In contrast, measurements on real hardware can be very accurate but cover only one design point. Simulation bridges the gap between the two other techniques by offering good accuracy while covering reasonably large design spaces. Unfortunately, the performance overhead of simulation is significant.

For single-thread applications, sampling-based techniques have proven to be very effective at reducing simulation time with a very small reduction in accuracy [10, 18, 23, 24]. The key idea is to identify a small set of samples to simulate and then infer the results for the complete application. The number of committed instructions is used to identify samples in a hardware independent manner. This metric is hard to adapt to a multi-threaded environment because of the existence of busy wait loops. Busy wait loops include instructions that do not result in the benchmark making forward progress. Alameldeen and Wood [1] recommend to use a work related metric for such situations.

Inspired by previous work [18], this paper is a necessary first step towards sampling-based

simulation of TBP applications. We first run a fast, low detail simulation and sample the execution to gather performance measurements. Next, we group together similar samples using a clustering algorithm [12] and select a representative point for each cluster. Finally, we create an estimation of the detailed execution by simulating only the representative samples.

In this work, we make two main contributions. First, we propose the number of completed tasks as a progress metric for sampling the TBP applications. This makes it possible to divide the total work of the application into equal size samples even with the existence of busy wait loops.

Sampling strategies commonly exploit the periodic behavior of applications. As a second contribution of this work, we show that our TBP applications present with this type of behavior. In fact, our TBP applications are highly periodic, needing only 5 samples on average to get an error below 5 %. In our experiments, 5 samples correspond to an average of 0.5 % of the total execution, when using a sample size of 1000 completed tasks. In addition, we investigate the effects of other implementation decisions like the initial centroid placement and how to choose representative points.

A long term goal is to use the sample information to simulate selected parts of the application with a high degree of detail. This approach may increase simulator performance significantly. To focus fully on the sampling methodology, we leave this promising application as further work.

The rest of this paper is organized as follows. Section 2 discusses the background for our work. Section 3 describes our proposed method for reducing the simulation time and Section 4 gives insight into our simulation framework. Section 5 presents our results and Section 6 concludes the paper.

B.2 Background

Simulation is a key technique for computer architecture research so there is a high motivation in developing ways to reduce the simulation time [24]. One way to overcome the problem of slow simulations is to trade accuracy for a faster execution. However, this is not possible or desirable in many cases.

Another simple strategy is to run only a portion of an application's execution and assume that this is representative for the whole execution. For applications where starting the execution at a specified point is not possible, a technique called *fast-forwarding* can be used [24]. When fast-forwarding, a simulator only emulates the hardware system allowing for a faster simulation until an interest point in the execution. Another option is to use *checkpointing*. When checkpointing, a snapshot of the simulated system state is stored, making it possible to restart execution from that point. The main disadvantage of this technique is the potentially large storage space required for the checkpoint.

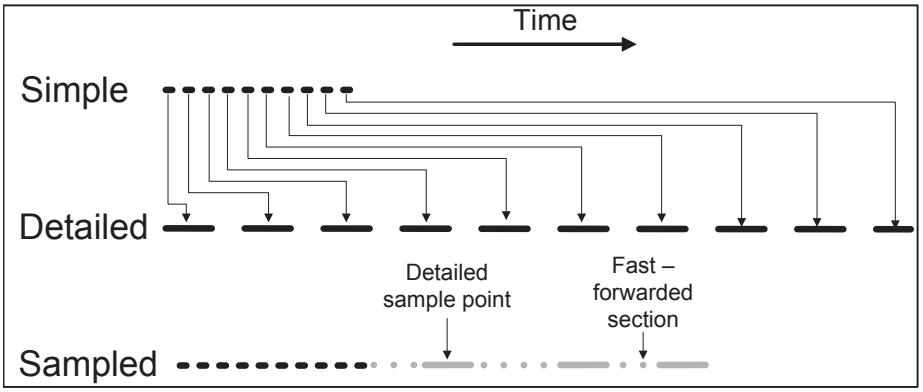


Figure B.1: Simulation times for Simple, Detailed and Sampled simulation modes

Algorithm 1 Simulation Methodology

```
Perform sampling in Simple simulation mode and create execution profiles
Group sample points into clusters using K-means
Choose a representative sample point for every cluster
while representative points still exists do
    fast-forward to the next representative sample point
    execute in detailed simulation mode
```

Perelman et al.[18] introduce SimPoint, a profile-driven clustering-based analysis tool which is capable of identifying a set of execution samples (called *simulation points*) that are independent of the hardware platform. An improved version of this tool, SimPoint 3.0, is described in [10]. The *simulation points* are representative for various repetitive parts of the program's execution. By simulating them in detail and then weighing their results according to the size of the program region they are part of, one can estimate the complete detailed execution with significant reduced simulation time.

SMARTS [23] employs statistical sampling to collect and execute in detail a large number of very small (in terms of number of instructions) execution samples. An important characteristic of SMARTS is that it allows to trade off confidence in results against speed of execution. Both SimPoint and SMARTS are designed for single-threaded applications and are used in conjunction with single-core simulation platforms.

There are also other approaches for accelerating simulations of parallel architectures. Rico et al.[20] argue that the level of abstraction used by most current simulators can be misleading for some studies like early processor design or high-level explorations. They introduce *TaskSim*, an architecture simulator designed for many-core platforms that can provide several levels of modeling abstraction. Another way to reduce simulation time is to distribute the simulation across multi-core or multi-machine environments. Graphite[15] and SlackSim[3] are two parallel simulation tools.

B.3 Sampling TBP applications

The 4 steps of our methodology are summarized in Algorithm 1. In this paper, we focus on the sampling, clustering and representative points selection steps of this methodology. Since all these steps use data from the fast, low detail simulation mode, we performed all simulations in this mode. We study the impact of several parameters on the accuracy of the sampled simulation. Each sampled simulation is compared against a complete execution in simple mode to determine its accuracy. We leave switching between 2 simulation modes (simple and detailed) as future work.

The complete methodology described by Algorithm 1 can be summarized as follows. In the first step, we sample the execution of the application and gather the information required to create a profile using the simple simulation mode. Next, we use the profile of the application to run a clustering algorithm in order to group the data points into K clusters. For each cluster, we calculate a centroid (the average of all the points in that cluster) and a weight (the number of data points in each cluster relative to the total number of points). The centroids are used in the next step to select the representative sample point for each cluster. The weights are used in the final step to create an estimate for the complete execution. As Figure B.1 shows, the total time required for the *sampled mode* is the time to run the simple simulation for profiling plus the time it takes to simulate the representative samples in detail and to fast-forward between them. All in all, a reduction in simulation time is achieved.

B.3.1 Choosing Samples

Sampling-based techniques aimed at single-thread applications use the number of committed instructions for sampling. This metric is not suited for parallel environments [1] because of busy wait loops. For this reason, we propose to use the number of completed tasks as a progress metric for TBP applications. In the sampling step of our technique, we count the number of completed tasks of all working threads collectively. For every s completed tasks, we record the simulated time which allows us to create the profile of the application.

B.3.2 Clustering Samples

In order to identify the repetitive sections in each execution, we employ the *K-means* algorithm to group similar data points into clusters using the simulated time as a metric. *K-means* [13] is a well known iterative clustering algorithm and Algorithm 2 illustrates its main steps. Clustering is done in two phases, which are repeated to convergence. The main parameter of the algorithm is the number of cluster K to be created. The placement of the initial centroids among the sampled data can also have an impact on the results. We investigate the accuracy impact of both of these parameters in Section B.5.

Algorithm 2 *K*-means algorithm

```
place K points (centroids) among the data points
while points change cluster membership do
    for all sample points do
        calculate the distance to each centroid
        assign the point according to the minimal distance
    for all centroids do
        new position = average of all the points in the cluster
```

The algorithm has converged when the sample points cease to change cluster membership between iterations. The output of this algorithm are the coordinates of the centroids, a set of weights for each cluster and a mapping of each data point to a cluster. Figure B.2 shows the profile of one of our benchmarks together with the clustering of the data points (for K = 6). The enlarged points are the centroids of each cluster.

B.3.3 Implementation parameters

In this paper, we study the sampling, clustering and representative points selection methods and their impact on the accuracy of the sampled simulation. We experiment with several parameters for these methods and we also reason about the impact on the potential speedup.

The first parameter we investigate is the sample size s. As mentioned before, we use the number of completed tasks as a progress metric for sampling. Each sample contains a total of s completed tasks, counted for all executing threads collectively. The value of s impacts the detection of the repetitive patterns in the execution. The relation between the value of s, the number of sample points (S) and the total number of tasks (T) is given by the equation: $S = T/s$. A low s value creates a more detailed profile, since more sample points are available. However, there are 2 disadvantages for sampling with a low s value: (I) it takes longer because of the larger overhead (sampling, dumping stats) and (II) below a certain s no useful information is added to the profile. In contrast, a high s value leads to a coarser profile, with larger but fewer sample points, which can miss patterns. Additionally, in the last step of Algorithm 1, larger sample points take longer to simulate in detail, which impacts the potential speedup.

The second parameter we investigate is the number of clusters K used by K-means. This parameter impacts the way the samples are grouped together and, similar to s, it can affect the potential speedup when simulating the representative sample points.

Third, we investigate 3 methods for placing the initial centroids for K-means:

- *Equally spread* — We spread the initial centroids evenly among the minimum and maximum values of the sample points.
- *Initialization Aware* — We define 3 centroids that are placed as follows: one is placed among the data points of the initialization phase of the application, one

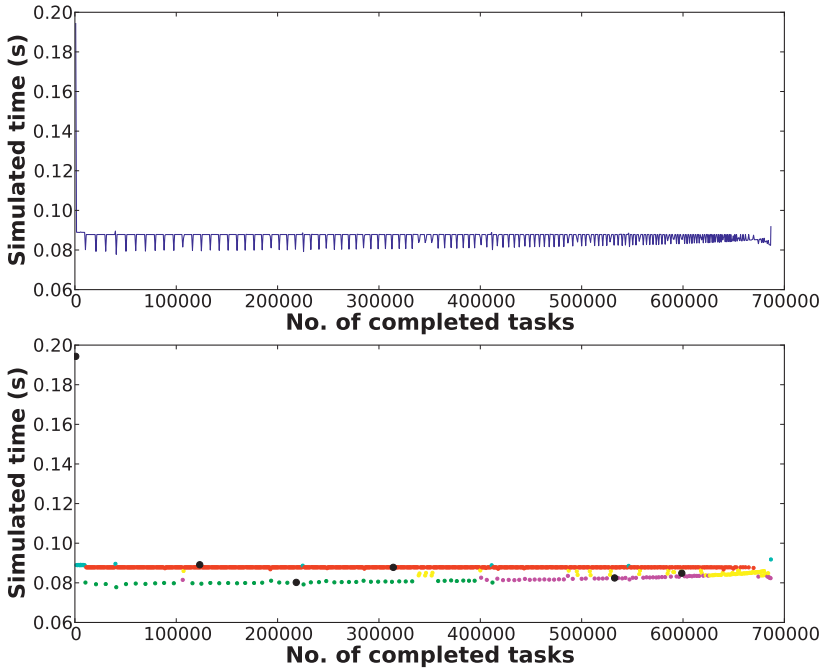


Figure B.2: SparseLU: profile (top) and clusters (bottom)

is placed among the minimum values of the whole data set and one among the maximum values of the whole data set. For $K > 3$, the extra centroids are evenly distributed among the minimum and maximum values of the data set.

- *Random*

The last parameter we investigate is the way the representative points are selected after the centroids are determined. We investigate 3 methods:

- *First after centroid* — The first sample point positioned after the centroid is selected.
- *First appearance* — The first sample point in each cluster is selected. This method generates the sampled simulation with the smallest fast-forwarding time.
- *Distance Threshold* — The first point whose value is greater or equal to a given percentage of the centroid's value is selected (we experimented with 80 %, 85 %, 90 % and 95 %).

Table B.1 BOTS input workloads

	FFT	Fib	nQueens	Sort	SparseLU	Strassen
Input	2^{25} floats	30'th element (no cut-off)	14x14 board	2^{25} integers	200x200 sparse matrix of 25x25 blocks	2048x2048 matrix

B.4 Methodology

We use the cycle-accurate GEM5 simulator [2] in full-system mode with a 2.6.27 customized Linux kernel. We simulate both the application and the operating system (OS) because TBP libraries rely on the OS to manage shared memory and provide a thread and process abstraction. The simulated platform is an in-order, 2-core processor with a clock frequency of 1GHz. We simulate a 2 level cache hierarchy with split data and instruction private cache (64 KB each) and a shared L2 cache (2 MB). Since this is an early study, we limit our experiments to a dual-core CPU to reduce simulator overhead. In future work we will simulate a platform with more cores. To be able to simulate the benchmarks to completion within reasonable time, we perform our simulations in GEM5's *atomic* mode. In atomic mode, only basic characteristics of the CPU are simulated.

In this work, we use a subset of the Barcelona OpenMP Task Benchmark Suite (BOTS)[5]. BOTS is a benchmark suite assembled to assess the performance of task-based programming models. Table C.1 lists the input workloads we used in our experiments. In these experiments, the default OpenMP parallelization was changed to a Wool implementation. Wool [7] is a lightweight, work-stealing TBP library that was designed with the ability to scale well with small tasks (smaller than a hundred cycles). It uses *pthreads* to create a number of working threads, each of them with its own task queue. The Wool library was customized to signal GEM5 every time a task is completed. A dedicated counter is incremented every time a signal is received from the application.

B.5 Results

In this work, we explore a large design space for sampling-based simulation of TBP applications. We experiment with values of s and K and the methods for initial placement of centroids as well as selecting the representative sample point. Unless otherwise stated, the results are presented for s = 800 tasks, K = 10 clusters, initial centroid positioning policy IA and *First after centroid* representative sample selection method.

As we mentioned in Section B.3, the sampled simulation is compared against a complete execution in simple mode. Performance is calculated as the total number of tasks executed divided by the total simulated time. The *Performance estimation error (%)* in Figure B.3, B.5, B.7 and B.8 refers to the difference (in percentage) between estimated performance

of the sampled simulation (P_{S}) and performance of the complete simple mode simulation (P_{Full}): *Performance estimation error (%)* $= (P_{\mathrm{S}} - P_{\mathrm{Full}})/P_{\mathrm{Full}} * 100$.

SparseLU registered highest accuracy in most of our experiments (see Figure B.3, B.5 and B.8). Figure B.2 illustrates why our scheme works very well for this benchmark. Here, the sample points within a cluster are very similar which makes estimation based on a representative sample point very accurate.

B.5.1 Sample Size

Figure B.3 shows the accuracy impact of the sample size *s*, for several values of *K*. For low values of *s*, the sample points contain little information. If too few of them are simulated (*K* is low also), the estimations based on them are inaccurate.

It is also interesting to observe that increasing *s* will not always lead to a smaller error. In Figure B.3a, FFT has a higher error value for 3000 tasks than 400 tasks. This is explained by the effect of *s* on the execution profile of the application. For high values of *s*, the execution profile ceases to capture all the trends and this impacts accuracy. This can be observed in Figure B.4.

B.5.2 Number of Clusters

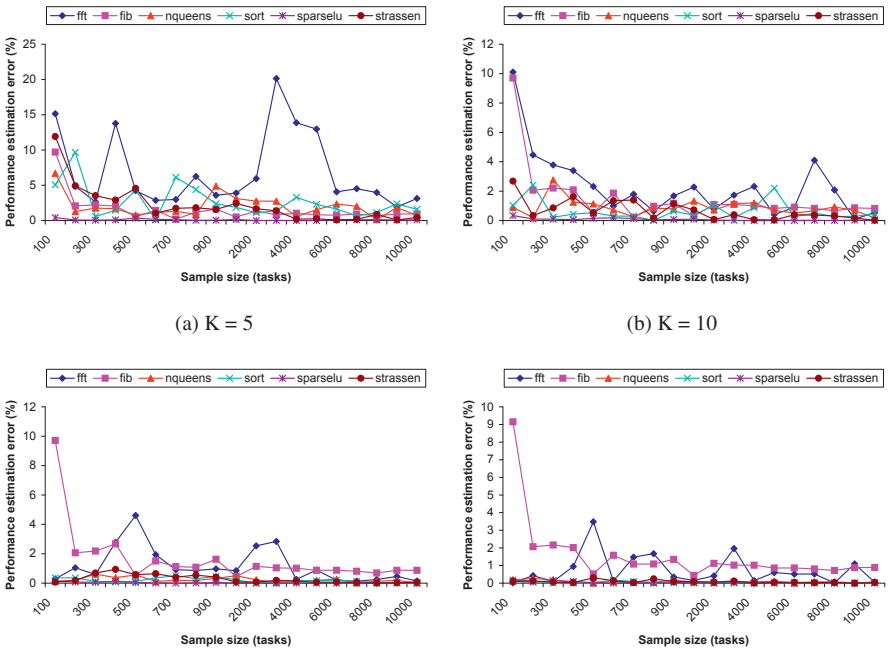
The number of clusters is the most important input parameter for the clustering algorithm. Balancing *K* and *s* allows us to produce accurate simulations. Figure B.5 shows how *K* impacts the accuracy of the simulation for different values of *s*.

As mentioned in Section B.3, the value of *K* also impacts the potential speedup of the simulation. Considering the number of tasks simulated in Sampled mode (*Tasks_in_Sampled=*$K \cdot s$) and the total number of samples in the full execution (*Total_samples*), we compute the *Fraction of execution=Tasks_in_Sampled/Total_samples*. Figure B.6 presents the *Fraction of execution* (as percentage) for 4 values of *s*. The values exceeding 100 % for nQueens, Sparselu and Strassen in Figure B.6d are due to the fact that these benchmarks execute less than 1 million tasks in total. Consequently, Sampled mode requires more samples ($K \cdot s = 1000000$) than are available.

B.5.3 Initial Position of the Centroids

The initial positioning of the centroids is the second input parameter of *K*-means. It impacts the final position of the centroids and ultimately the selected representative sample. Figure B.7 shows the accuracy of the three methods presented in Section B.3.

We investigated the *Initialization Aware* positioning policy in order to better model the initialization phase of the tested benchmarks. A good candidate for this policy is Sort



(a) K = 5

(b) K = 10

(c) K = 25

(d) K = 50

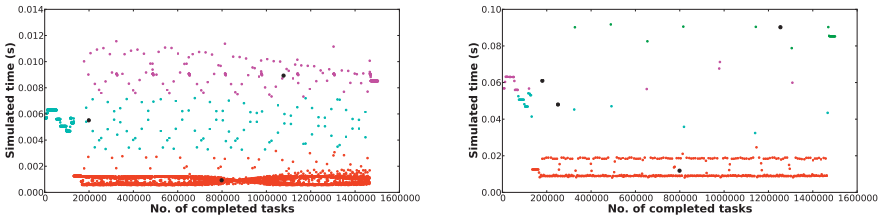
Figure B.3: Sample size vs. simulation accuracy

which has an initialization phase of about 30 % of the total execution. In Figure B.7, Sort has an error below 0.01 % for this policy. For low values of s and K, IA shows better accuracy results for Sort, but also for Fib and Sparselu. All 3 policies have very similar accuracy values for high values of K.

B.5.4 Selecting the Representative Points

Representative point selection impacts both accuracy and simulation speed. Figure B.8 presents the impact on accuracy for all methods described in Section B.3. The average results in Figure B.8 show that the *First appearance* selection yields the highest error while the *First after centroid* the smallest. However, there are some rare situations where the value of the first point of a cluster is very close to the one of the centroid. By selecting this point as representative, we get a lower error than by using the *Distance Threshold* methods. This can be observed in Figure B.8 for nQueens and Strassen.

The selection of the representative points is important because the fast-forwarding time in Sampled mode can be reduced by choosing earlier samples in the execution. Table B.2



(a) s = 400 tasks (b) s = 3000 tasks

Figure B.4: Sample size vs. profile accuracy

Table B.2 Simulation time for Simple, Detailed and Sampled modes

	Measurements				Estimations		
	$\overline{T}_{Simp}$ (s / 1000t)	$\overline{T}_{Det}$ (s / 1000t)	T_{Simp} (s)	S	T_{Det} (s)	T_{Samp} (s)	Speedup
fft	7.40	134.27	11124.3	1503	201804.53	22882.64	8.82 x
fib	0.042	0.56	57.04	1347	749.04	116.62	6.42 x
nqueens	117.89	1523.32	100085.89	849	1291857.19	207087.53	6.24 x
sort	4.10	147.06	5501.24	1343	197362.46	11713.42	16.85 x
sparselu	99.62	1225.66	68436.89	687	841663.52	142474.13	5.90 x
strassen	12.14	167.36	10005.19	824	137831.84	20780.93	6.63 x

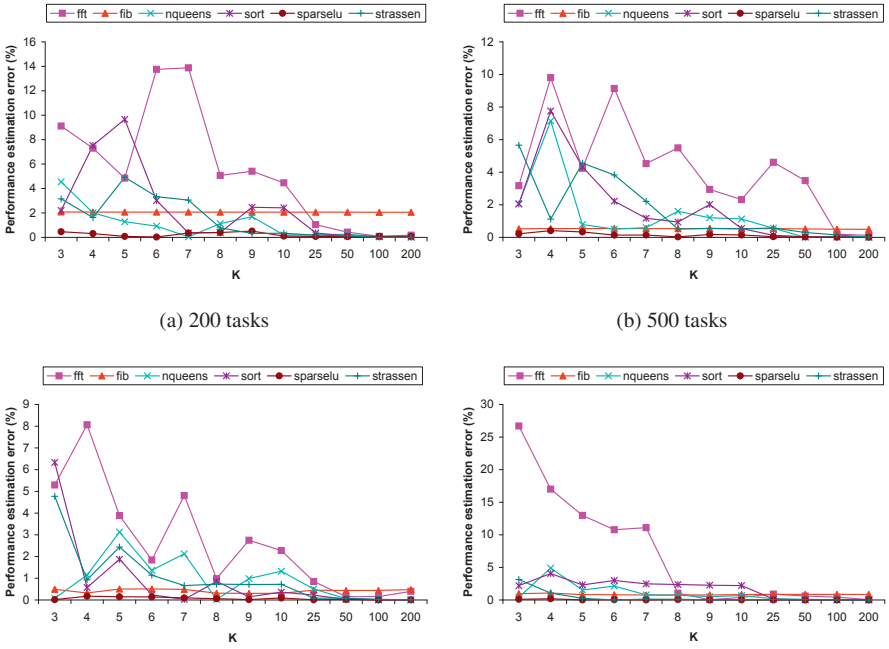
lists the measured average time for executing 1000 tasks and the total simulation time in Simple mode. We also simulated a detailed execution of each benchmark for 3000 completed tasks. We averaged the time for executing 1000 tasks ($\overline{T}_{Det}$) and estimated the full execution time ($T_{Det} = \overline{T}_{Det} \cdot S$) with the total number of samples (S). The table also presents the estimations for Sampled mode according to Equation B.1.

$$T_{Samp} = T_{Simp} + \overline{T}_{Simp} \cdot (S - K) + \overline{T}_{Det} \cdot K \tag{B.1}$$

The Sampled mode simulation time is estimated for the worst case scenario where the last representative sample point is the last sample point in the execution. In this scenario, the fast-forwarding time is at its maximum value. We also calculate the potential speedup: $Speedup = T_{Det}/T_{Samp}$. For these measurements we used $s = 1000$ completed tasks. In Equation B.1, T_{Samp} assumes K = 5 clusters.

B.6 Conclusion and Future work

This work is a necessary first step towards efficient simulation of TBP applications. We introduce the number of completed tasks as a work-related metric that can be used to divide the execution of a TBP application into fixed-size samples. Furthermore, we show that our TBP applications exhibit periodic behavior, an average of 5 samples being sufficient to achieve an accuracy below 5 %. This observation has a significant potential



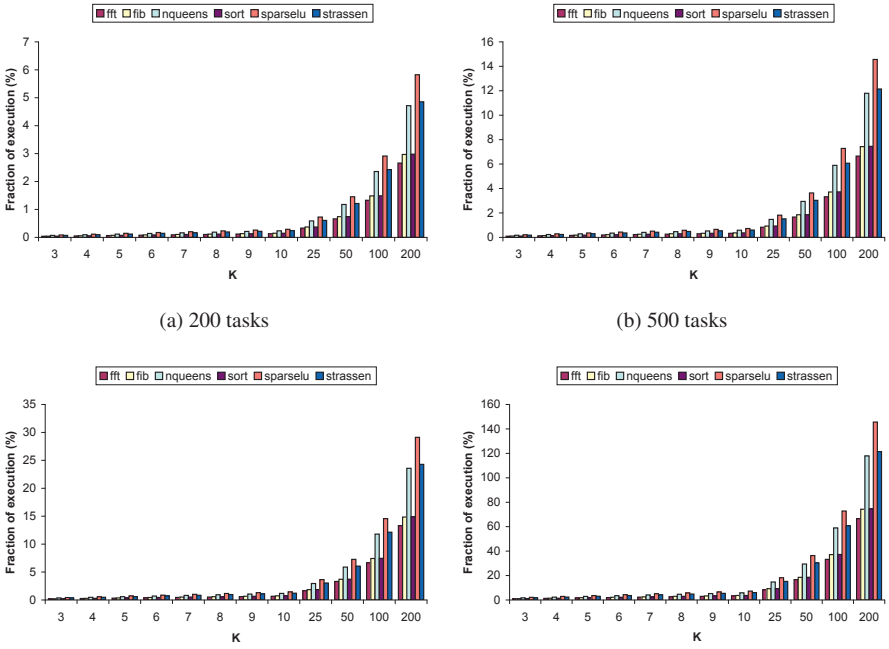
(a) 200 tasks

(b) 500 tasks

(c) 1000 tasks

(d) 5000 tasks

Figure B.5: Number of clusters (K) vs. simulation accuracy



(a) 200 tasks

(b) 500 tasks

(c) 1000 tasks

(d) 5000 tasks

Figure B.6: Fraction of execution

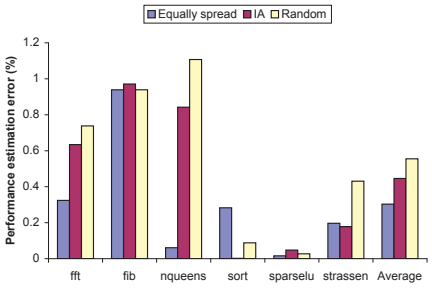


Figure B.7: Initial centroid position policy

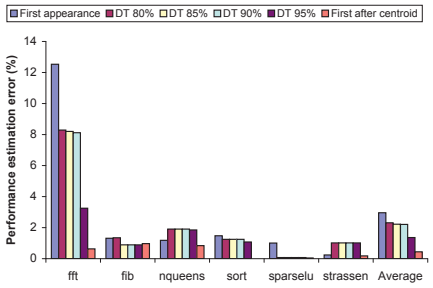


Figure B.8: Representative sample selection

for accelerating simulation since only 0.5 % of the application needs to be simulated to compute an accurate estimate of application performance.

Our plan for further work is to leverage the insight obtained in this work to improve simulator performance for TBP applications. There are at least two possible implementation strategies. In the first scheme, we start by simulating the application with a low degree of accuracy to chose the representative points. Then, we gather detailed simulation results from these samples. The main advantage of this scheme is that it is robust to changes in the architecture and benchmarks since we create a new profile with every new architectural configuration. The second scheme assumes that it is possible to identify characteristics of a task that are independent of the computer architecture. In this case, we can profile the execution once and reuse the profile for all architectural configurations. This scheme amortizes the overhead of sampling over many simulations. For both schemes, it is possible to use fast-forwarding between the detailed sample points or to create checkpoints.

Bibliography

[1] A. R. Alameldeen and D. A. Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17, July 2006. ISSN 0272-1732. doi: 10.1109/MM.2006.73. URL `http://dx.doi.org/10.1109/MM.2006.73`.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The GEM5 Simulator. *SIGARCH Comput. Archit. News*, 39, 2011.

[3] J. Chen, M. Annavaram, and M. Dubois. SlackSim: a platform for parallel simulations of CMPs on CMPs. *SIGARCH Comput. Archit. News*, 37, 2009.

[4] Cilk++. Cilk++: A quick, easy and reliable way to improve threaded per-

formance. Available: http://software.intel.com/en-us/articles/intel-cilk-plus/. (retrieved November 2016).

[5] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing*, pages 124–131, Sept 2009. doi: 10.1109/ICPP.2009.64.

[6] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21, 2011.

[7] K.-F. Faxén. Wool - A Work Stealing Library. *SIGARCH Comput. Archit. News*, 36(5):93–100, June 2009. ISSN 0163-5964. doi: 10.1145/1556444.1556457. URL http://doi.acm.org/10.1145/1556444.1556457.

[8] S. Fuller and L. Millett. Computing Performance: Game Over or Next Level? *Computer*, 44(1):31–38, Jan 2011. ISSN 0018-9162. doi: 10.1109/MC.2011.15.

[9] R. L. Graham. Bounds of Certain Multiprocessing Anomalies. *The Bell System Technical Journal*, XLV:1563–1581, November 1966.

[10] G. Hamerly, E. Perelman, J. Lau, and B. Calder. Simpoint 3.0: Faster and more flexible program analysis. In *Journal of Instruction Level Parallelism*, 2005.

[11] D. Leijen, W. Schulte, and S. Burckhardt. The Design of a Task Parallel Library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640106. URL http://doi.acm.org/10.1145/1640089.1640106.

[12] D. J. MacKay. *Information Theory, Inference and Learning Algorithms.Chapter 20: An Example Inference Task: Clustering*. Cambridge University Press, 2003.

[13] MacQueen, J.B. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, California, 1967. University of California Press. URL http://projecteuclid.org/euclid.bsmsp/1200512992.

[14] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert. Frameworks for multi-core architectures: a comprehensive evaluation using 2D/3D image registration. In *Proceedings of the 24th International Conference on Architecture of Computing Systems*, 2011.

[15] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In

HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture, pages 1–12, Jan 2010. doi: 10.1109/HPCA.2010.5416635.

[16] K. Olukotun and L. Hammond. The Future of Microprocessors. *Queue*, 3(7):26–29, Sept. 2005. ISSN 1542-7730. doi: 10.1145/1095408.1095418. URL http://doi.acm.org/10.1145/1095408.1095418.

[17] D. Patterson. The Trouble With Multicore. *IEEE Spectrum*, 47(7), 2010. ISSN 0018-9235.

[18] E. Perelman, G. Hamerly, and B. Calder. Picking Statistically Valid and Early Simulation Points. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, pages 244–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2021-9. URL http://dl.acm.org/citation.cfm?id=942806.943854.

[19] J. Perez, R. Badia, and J. Labarta. A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures. In *IEEE International Conference on Cluster Computing*, 2008.

[20] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramirez, and M. Valero. On the simulation of large-scale architectures using multiple application abstraction levels. *ACM Trans. Archit. Code Optim.*, 8(4), 2012.

[21] K. Skadron, M. Martonosi, D. I. August, M. D. Hill, D. J. Lilja, and V. S. Pai. Challenges in Computer Architecture Evaluation. *Computer*, 36(8):30–36, Aug. 2003. ISSN 0018-9162. doi: 10.1109/MC.2003.1220579. URL http://dx.doi.org/10.1109/MC.2003.1220579.

[22] TBB. Intel Threading Building Blocks. http://software.intel.com/sites/products/documentation/hpc/tbb/getting_started.pdf. (retrieved November 2016).

[23] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Statistical Sampling of Microarchitecture Simulation. *ACM Trans. Model. Comput. Simul.*, 16(3):197–224, July 2006. ISSN 1049-3301. doi: 10.1145/1147224.1147225. URL http://doi.acm.org/10.1145/1147224.1147225.

[24] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and Comparing Prevailing Simulation Techniques. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 266–277, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2275-0. doi: 10.1109/HPCA.2005.8. URL http://dx.doi.org/10.1109/HPCA.2005.8.

Appendix C

Paper B.2

Challenges of reducing cycle-accurate simulation time for TBP applications

Alexandru C. Iordan, Magnus Jahre and Lasse Natvig

International Conference on Computational Science (ICCS)
2013

Abstract

Cycle-accurate simulation is an important tool that depends on the computational power of supercomputers. Unfortunately, simulations of modern multi-core platforms can take weeks or months. In this paper, we look into the challenges of employing a sampling based technique for reducing simulation time of multi-threaded applications. We introduce FASTA, a simple 3-phase methodology for reducing the simulation time of Task Based Parallel applications. FASTA takes advantage of the periodic behavior of parallel applications and identifies a small number of representative execution samples. By exploring a large design space we show that even though we can not use FASTA for every type of application, there are some for which a 12x speedup can be achieved with an accuracy error as low as 2.6%.

C.1 Introduction

Increasing in complexity with each generation, developments in hardware and software design require ten of thousands of computational hours for simulation and testing. For computer architects simulation is also an indispensable technique for exploring novel ideas. The main advantages of simulation are increased flexibility for design space exploration and outputs with extensive, noninvasive and detailed measurements. In this way, simulation bridges the gap between analytical modeling and real world measurements. Unfortunately, the cost of these advantages is long simulation times.

In today's ICT market, Chip Multiprocessors (CMPs) or multi-core architectures are the platform of choice in almost all segments. CMPs were first introduced as a solution to a design constraint known as the *power wall*. For decades, CPU designers took advantage of the increasing transistor count on a chip to boost computational power. They did this by increasing the chip's clock frequency and by exploiting instruction level parallelism (ILP) more aggressively. However, cooling system limitations and diminishing returns from ILP created the need for a new architectural approach. CMPs utilize the hardware resources provided by production technology improvements to create several less complex cores. This architecture is able to exploit both instruction and thread level parallelism allowing for an aggregate performance increase. At the same time, they require a lower power budget [10].

The switch to the multi-core model has placed a new burden on programmers. CMPs can not be fully exploited using sequential programming and parallelization is required to achieve maximum performance. For this reason, there is a considerable interest in programming models that can simplify parallel programming. In this work, we focus on *Task Based Programming* (TBP), a parallel programming model that has received significant attention recently [6, 9, 12, 17]. TBP's key idea is that the programmer partitions the application into tasks and specifies the dependencies between them [9]. A task is a light weight unit of work, and all independent tasks can be executed concurrently. A runtime task management library distributes tasks to processing units and enforces all dependencies.

There have been many attempts to reduce simulation time. For single-core simulation, sampling-based techniques are very effective at reducing simulation time with a small loss in accuracy. SimPoint tries to identify a set of samples that are representative for the entire execution [15]. The complete execution is inferred from simulating and weighing only these samples thus reducing the total simulation time. In a different approach, SMARTS uses statistical sampling theory and samples the execution at fixed interval [20]. A more detailed study of simulation time reduction methods can be found in [22] showing that sampling achieves high accuracy for single threaded applications.

There are two major issues when sampling multi-threaded applications. First, we have to use a metric that is proportional to forward progress of execution. Single-threaded applications are sampled using metrics like the committed number of instructions or IPC/CPI.

As argued by Alameldeen and Wood, these performance metrics are not suited for parallel executions [1]. For example, while executing a busy wait loop, the instruction count of the thread will increase but the execution will not move forward. Second, we need to account for the interleaving of threads in different executions. Because of shared resources and data dependencies, different runs of the same parallel application can have different instruction or task streams.

In this paper, we use a 3-phase approach called FASTA that aims to reduce simulation time of TBP applications running on CMP platforms. First, we run a fast, low detail simulation to sample the execution and gather measurements. Next, we use a clustering algorithm to group together similar samples and to select a representative point for each group. Finally, we simulate the representative samples in detail and we use the results to estimate the full detailed execution. Through our experiments we found that the thread interleaving impacts the accuracy of the FASTA results more for some classes of parallel applications than others. This impact becomes more prominent with the increase of core count. However, we found a class of parallel applications that are not affected by the interleaving problem. In these cases, FASTA results show an average error below 4% and a speedup of 12x maximum.

C.2 Background

Researchers have been motivated to find ways to reduce simulation time due to simulation's key role for software and hardware development [22]. One proposed technique is to reduce the input set of the benchmarks [2, 11]. This approach assumes that the characteristics of the full input set can be preserved even though fewer instructions are actually executed. However, the accuracy of this approach is generally poor [22].

Another technique for addressing long simulation time is truncated execution. In this approach, a continuous section of X instructions is selected and simulated in detail. If this section is not in the beginning of the application, *fast-forwarding* or *checkpointing* can be used to start the simulation at the desired point. When fast-forwarding, a simulator only emulates the hardware system up to the desired execution point. With checkpointing, a snapshot of the simulated system state is stored, allowing for a restart from that point.

In single-threaded environments, sampling-based techniques have been used to reduce simulation time while maintaining good accuracy. There are primarily 3 classes of sampling techniques [21]:

- *Representative sampling techniques* attempt to identify a sample or a group of samples in the simulated code that can be held representative for the entire execution. Perelman et al. [15] use the K-means algorithm to group similar simulation points into clusters and then calculate a centroid for each cluster. The "closest" sample point to the centroid is considered representative for the entire cluster.

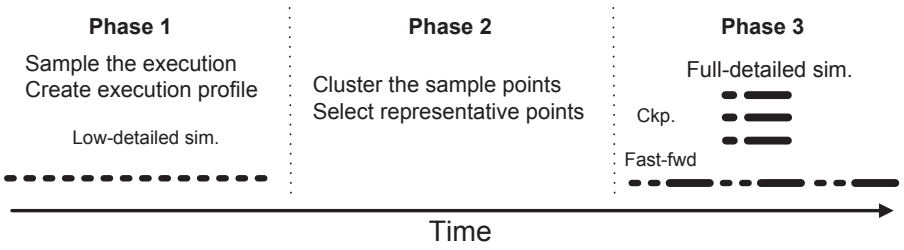


Figure C.1: The FASTA flow

- *Periodic sampling techniques* select portions of the simulated code at periodic intervals. SMARTS [20] samples a large number of very small (in terms of number of instructions) execution points which are later simulated in detail. An important characteristic of SMARTS is that it allows to trade off result confidence against speed of execution. Building on their work with SMARTS, Wenisch et al. developed another statistical sampling methodology called SimFlex [19]. Using a different approach for warming the CPU units, SimFlex reduces the total simulation time. In addition, SimFlex can be employed for multiprocessor throughput applications like those in the TPC-C OLTP or Specweb99 benchmark suites.

- *Random sampling techniques* try to combine the results from N randomly selected simulation points to produce the overall result. To improve the accuracy of this technique Conte et al. [7] suggest the use of longer warm-up periods and increasing the number of instructions in each sample.

For very large input sets even functional simulation (i.e. fast-forwarding) is too time consuming. In such cases, a technique called *direct-execution* can be employed. The host machine is used to execute the application and checkpoint the execution. This checkpoint is then transferred to the simulator. In order for this technique to work, the host and simulated machines must use the same ISA or cross-compilation needs to be employed before the transfer.

Simulation can be also accelerated by varying the abstraction level or by exploiting parallel hardware. Rico et al. [16] argue that the level of abstraction used by most current simulators can be misleading for some studies like early processor design or high-level explorations. They introduce *TaskSim*, an architecture simulator designed for many-core platforms that can provide several levels of modeling abstraction. Another way to reduce simulation time is to distribute the simulation across multi-core or multi-machine environments. Graphite[14] and SlackSim[5] are two parallel simulation tools.

C.3 FASTA

In this paper, we propose a simple methodology for *representative sampling*, called *FAst Simulation of TBP Applications* (FASTA). Like Perelman et al. [15], our technique exploits the periodic behavior of parallel applications and identifies representative sections for each interval of the execution. There are 3 main differences between our approach and SimPoint: (1) we target the simulation of parallel application thus (2) we use a different work-related progress metric than the traditional CPI/IPC and (3) our methodology is not architecture independent.

Fig. C.1 gives an overview of the flow of our approach. To better understand how FASTA works and how different parameters can affect the results, we summarize each phase in the next subsections.

C.3.1 Phase 1: Sampling

We gather the simulated execution time for every s completed tasks and create an execution profile. Section C.4 discusses in detail why we use the number of completed tasks as a metric in our work. The value of s has an impact on the granularity of the profile: a low value will lead to a more detailed profile while a high value to a coarser one. A trade off is needed because simulating with a low s value requires more time while a coarser profile can miss patterns in the execution.

C.3.2 Phase 2: Clustering and Representative Point Selection

In Phase 2, we use the *K-means* algorithm on the resulting sample population. K-means is an well known iterative clustering algorithm [13]. The main input parameter for this algorithm is the number of cluster to be created (K). K affects both the accuracy and the potential speedup of the FASTA simulation.

The algorithm outputs (1) the coordinates of the centroids, (2) a set of weights for each cluster and (3) a mapping of each data point to a cluster. The "closest" sample point to its respective centroid is selected as representative for each cluster.

C.3.3 Phase 3: Detailed Simulation

In Phase 3, we start the full-detailed execution of the representative samples using the checkpoints or fast-forwarding. When using checkpoints, we can start the simulations concurrently because each checkpoint is independent of the others.

In the fast-forwarding approach, we emulate the execution up to the representative sample point where we switch to full-detailed simulation. Then, we simulate the representative

sample and dump the measurements before we fast-forward to the next representative point. However, switching back and forth between simple and full-detailed mode adds an overhead that affects speedup. Also, since some buffers are cleared when switching from detailed to simple simulation (TLBs for example) accuracy can also be impacted. For both approaches, the detailed results are weighted according to the output of K-means and an estimate of the complete execution in detailed mode is created.

We experimented with 3 different methods to generate the checkpoints:

- *Checkpoint Representative Sample Points (CRSP)*: After the representative sample points are selected, a second low-detail simulation is started and checkpoints are created only for the representative sample points (see Fig. C.1).
- *Checkpoint All Sample Points (CASP)*: Generate a checkpoint for each sample point during Phase 1.
- *Checkpoint at Intervals (CI)*: This approach is a trade-off between CRSP and CASP. During Phase 1, checkpoints are generated every X sample points, where X can be defined by the user. After selecting the representative sample points, the nearest preceding checkpoint is used to restart the simulation and create a new checkpoint for the representative sample.

All the checkpoints have been created using the built-in functionality of our simulator. We did not try to improve in any way this functionality, neither for storage neither for speed. Both [18] and [3] present ways in which checkpoint size and restoration time can be reduced. We consider that such work, though interesting and important in the simulation field, was beyond the scope of our research. We kept focus on validating our methodology rather than trying to optimize it.

C.4 The challenges

There are 2 main challenges to be addressed when employing sampling for reducing the simulation time of multi-threaded applications:

- sampling using a metric that is proportional to forward progress of execution
- the change in the instruction stream of a parallel application from one execution to another as races resolve differently on different runs

To address the first issue, we use *the number of completed tasks* as a work-related progress metric. We count the tasks collectively for all working threads. In this way we eliminate the problem of spin locks being recorded as false progress of the execution.

The second issue is much more difficult to deal with. Wenisch et al. conclude that it is unclear how to sample general multiprocessor applications so they focus their SimFlex methodology on throughput parallel applications [19]. Because in our approach we are trying to estimate the results of a detailed simulation using results from a very simple one,

Table C.1 BOTS input workloads

	FFT	Fib	nQueens	Sort	SparseLU	Strassen
Input	2^{25} floats	30'th element (no cut-off)	14x14 board	2^{25} integers	200x200 sparse matrix of 25x25 blocks	2048x2048 matrix

we need to assess the variability of the thread interleaving. To do that, we have experimented with 2 low detailed simulation modes. The first one models a simple CPU with instantaneous access to memory (Atomic mode) while the second one models the same CPU but with delays in accessing the memory system (Timing mode). By comparing the Atomic and Timing execution profiles, we can reason about the impact the memory model has on interleaving the threads. We also assess how the difference in CPU model affects the profiles, by comparing the results of the 2 low detailed simulation modes with the detailed ones.

C.5 Methodology

We use the cycle-accurate GEM5 simulator [4] in full-system mode with the 2.6.27 Linux kernel. We simulate both the application and the operating system (OS) because TBP libraries rely on the OS to manage shared memory and provide a thread and process abstraction. Our simulated platforms use 2-, 4- and 8-core CPUs, with a clock frequency of 1GHz and a 2 level cache hierarchy with a split L1 private cache (64 KB) and a shared L2 (2 MB). As mentioned before, for Phase 1 we have experimented with 2 simulation modes. The *Atomic* mode models a simple 5-stage pipeline In-Order CPU that can accesses the memory hierarchy without any resource contention or queuing delay. The *Timing* mode models the same simple CPU but its access to memory is realistic and includes delays. The simulations in Phase 3 are performed using a detailed Out-of-order CPU model.

In this work, we use a subset of the Barcelona OpenMP Task Benchmark Suite (BOTS) [8]. We changed the default OpenMP parallelization to a Wool [9] implementation. We also customized this library to signal GEM5 every time a task is completed. In this way, the simulator is able to keep record of each completed task and to implement the sampling for Phase 1. Table C.1 lists the benchmark input sets we used in our experiments. Using these inputs, the full detail simulation of some benchmarks running on the 8-core platforms lasted more than 18 days. For this reason we decided not to increase input sets with the core count.

C.6 Results

In the next subsections we present our accuracy, speedup and parameter variation results. Our design space is defined on 3 parameters: the sample size (ranging from s = 500 to s = 2000 samples), the number of clusters (ranging from K = 4 to K = 10 clusters) and the core count (2-, 4- and 8-core systems). We investigate the impact the interleaving of threads has on profiling the applications and ultimately on the accuracy of the FASTA results. We experiment with both checkpointing and fast-forwarding in order to determine the best speedup that we can achieve for our simulations.

C.6.1 Accuracy and speedup results

In order to quantify the error of the FASTA results we define the *performance* as the total number of tasks divided by estimated simulated time. We use the complete full-detail simulation of each benchmark as a baseline. The accuracy error is calculated as $\mathrm{E} = (P_{\mathrm{FASTA}} - P_{\mathrm{Full}})/P_{\mathrm{Full}}$ where P_{FASTA} is the estimated performance of the FASTA simulation and P_{Full} is the performance of the baseline.

Fig. C.2 presents the accuracy results for all benchmarks for s = 1000 completed tasks and K = 6 clusters across all test platforms. More results on the full ranges of both s and K are presented in Section C.6.2. As the average results show, the accuracy tends to decrease from 2- to 8-core systems. This trend can be partly explained by the fact that we did not increase our workloads with the number of cores. Because of this, the clusters are less differentiated for 8 cores, they can merge together and the selection of representative samples is erroneous. SparseLU and Strassen are affected by this type of errors. However, higher errors (like those of FFT and Fib) are due to the way threads interleave on different runs, this being discussed further in Section C.6.3.

As mentioned in Section C.3.3, using the fast-forward approach during Phase 3 can impact the accuracy of the results, not only the speedup. This is a short-coming of our experimental framework and not a limitation of the methodology. However, the clearing of buffers when switching back and forth between simulation modes is not the only cause for error. Thread interleaving has an impact here as well and is discussed in more details in Section C.6.3.

Fig. C.3 presents the potential speedups that can be achieved by using FASTA when compared to a complete full detailed execution of the benchmarks. These results are calculated for s = 1000 completed tasks and K = 6 clusters. Based on our results, we can conclude that for benchmarks with large input sets like FFT, Sort and Strassen, the checkpointing overhead is larger so CASP takes longer than fast-forwarding. If we use CRSP or CI then it will be generally faster than fast-forwarding. For Fib, nQueens and SparseLU checkpointing is always faster than fast-forwarding. It is also worth mentioning that checkpointing methods require storage space. FFT, Sort and Strassen need on average 156 GB, 176 GB and 115 GB respectively for a single simulation using the CASP method.

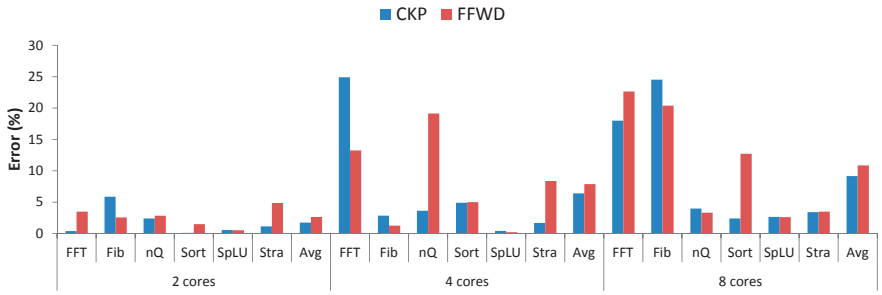


Figure C.2: The accuracy of FASTA simulations

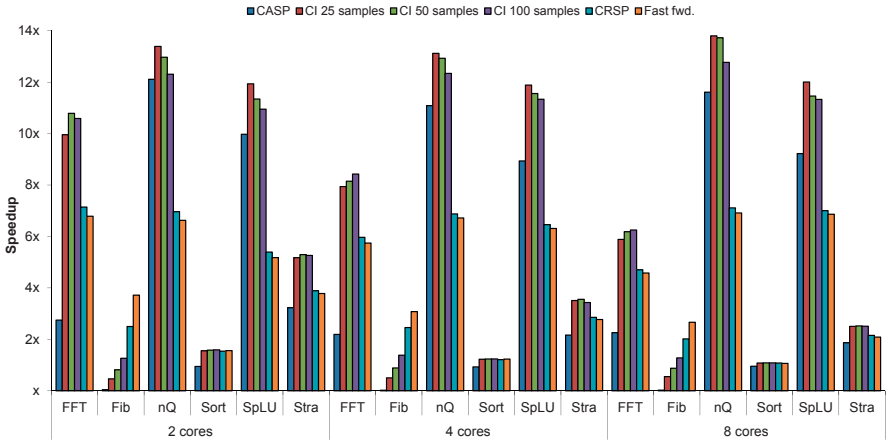


Figure C.3: The speedup of FASTA simulations

C.6.2 Parameter analysis

Fig. C.4 — C.6 show part of our results for the parameter study. Because of space limitation, we only present 2 sets of results for each test platform: the variation of K for s = 1000 and the variation of s for K = 6.

For reasons discussed in Section C.6.3, it is very hard to clearly quantify the effect of s or K on the accuracy of FASTA. However, for our simulations, we observed that a profile granularity of 800 - 1000 points gives the best accuracy results. This means, that knowing the total number of tasks an application will execute, we could calculate the value of s.

For a low value K, several patterns in the profile merge together in a cluster which results in a poor estimation. Generally, the higher the K the better the FASTA estimation is. However, we need to consider that K determines the number of samples that are simulated in detail during Phase 3. If we simulate too many samples during this phase, the potential speedup of FASTA decreases. So the idea is to select a low as possible K without affecting the accuracy of the estimation. In our experiments a value of 5 or 6 for K would yield a good accuracy level.

C.6.3 Thread interleaving

Given a random sample point in an application's profile, we were expecting to see different values of the simulated execution time when we simulate that application in different modes (Atomic, Timing or Detailed). What we also found was that a sample point will not contain the same work (the same tasks) across the 3 simulation modes. This can be seen as a "shift" of the profile spikes in Fig. C.7. Because of this "shift", the clusters and representative points selected in Phase 2 from a low-detail simulation profile do not map onto the Detailed profile. This behavior causes the FASTA estimations to be erroneous. This happens because threads do not interleave the same way for different simulation modes.

FFT calculates the discrete Fourier transform using the Cooley-Turkey algorithm and its execution is divided in 3 main phases. Because of the recursive approach used, tasks form a binary tree in each of the algorithm's phases. As the execution progresses, all sample points end up containing tasks from each of the 3 phases. The "shift" visible in Fig. C.7 is caused by races in accessing the data and the way the dependencies among tasks are handled. The estimation error varies from an average of 7.7% in the 2-core simulations to an average of 20.2% in the 8-core ones.

Fib, nQueens, Sort and Strassen are also implemented recursively and show the same behavior. However, how prominent the profile "shift" is differ from benchmark to benchmark. We found that the form of the application's task tree is closely correlated with the "shift". Fib and FFT have a binary task tree (2 children for every parent) and they are the ones affected by the largest profile "shift". Sort generates a task tree with 4 children for every parent and it is far less affected than the previous two benchmarks. For Strassen

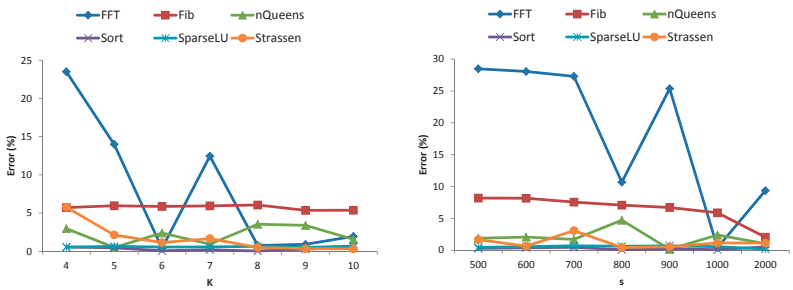


Figure C.4: The parameter analysis (K and s) - 2 cores

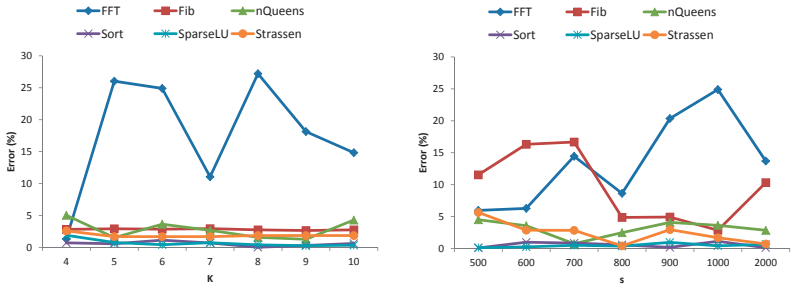


Figure C.5: The parameter analysis (K and s) - 4 cores

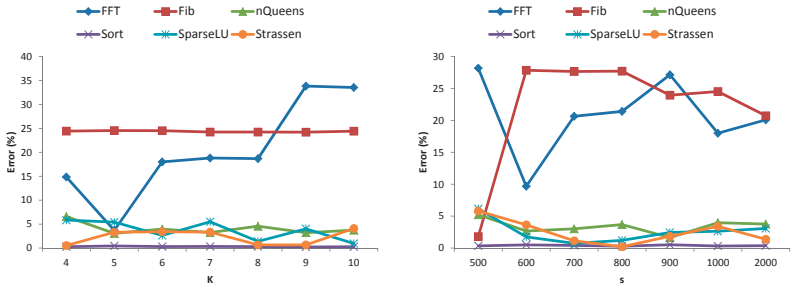


Figure C.6: The parameter analysis (K and s) - 8 cores

the "shift" is even less significant since its task tree has 7 children for every parent. The number of parent tree-node translates into the number of situations when the task stream can change. The higher the number of parent nodes, the higher the chances that race conditions will determine a different task stream for different executions.

In the case of benchmarks with more than 1 type of tasks (FFT and Sort), resource contention is another cause of changes in the task stream. This can be seen by analyzing an Atomic mode profile against a Timing mode one. Since the two simulation modes use the same simple CPU model, the profile "shift" is caused only by races in accessing the different memory models. By overlapping Timing mode profiles and Detailed mode ones, we could see yet another type of resource contention: the ones for CPU resources. Due to space limitation we could not include any such figures in this paper.

We also observed that the profile "shift" becomes more significant with the core count. In the 2-core experiments we recorded no such behavior, while for the 4-core ones profiles differed in some cases. The 8-core profiles are affected the most. This is due the fact that the number of races for resources (software or hardware) increase with the number of worker threads. So the number of situation when the task stream can change is greater for a higher core count.

There is 1 benchmark that was not affected by the profile "shift" (see Fig. C.8). *SparseLU* calculates the LU matrix factorization. The algorithm divides the input array into smaller blocks on which computation is performed. This is not a recursive algorithm and only 1 thread spawns all tasks. Because there is only 1 parent task and all children tasks perform the same work, race conditions do not change the task stream of this benchmark's execution. The average estimation error ranges from 0.5% for the 2-core test system to 3.7% for the 8-core one.

The fast-forwarding approach in Phase 3 is also affected by the thread interleaving problem, even though it is technically a single execution. If an application has a deep task tree (like FFT or Fib), then switching back and forth between 2 simulation modes will cause a similar "shift" of the representative samples as the one seen in Fig. C.7. In the case of SparseLU, using fast-forwarding is almost as accurate as using checkpointing (see Fig. C.2).

C.7 Conclusion

In this paper, we have introduced FASTA, a simple methodology for reducing simulation time of TBP applications running on multi-core platforms. In a 3-phase approach, FASTA samples and profiles the execution (Phase 1), identifies representative sample points through clustering (Phase 2) and simulates in detail the representative points (Phase 3). The results in Phase 3 are weighted and an estimate of the full-detail execution is created.

We investigated the challenges of using a sampling based methodology in a multi-threaded

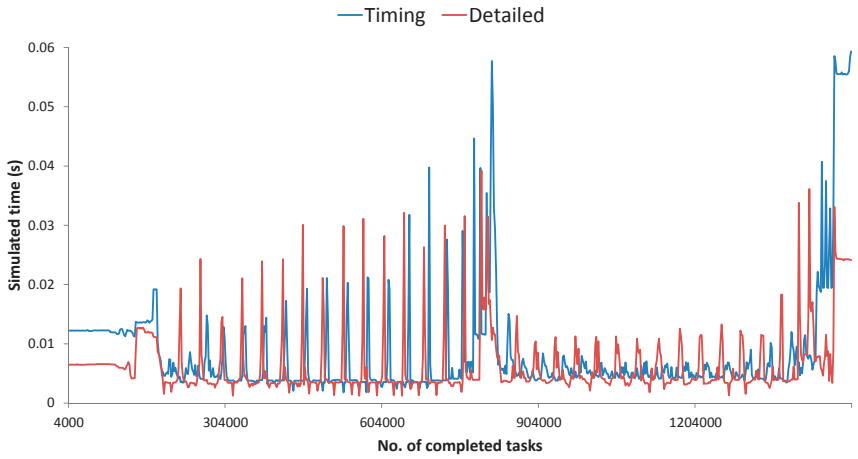


Figure C.7: FFT - Timing mode profile vs. Detailed mode profile

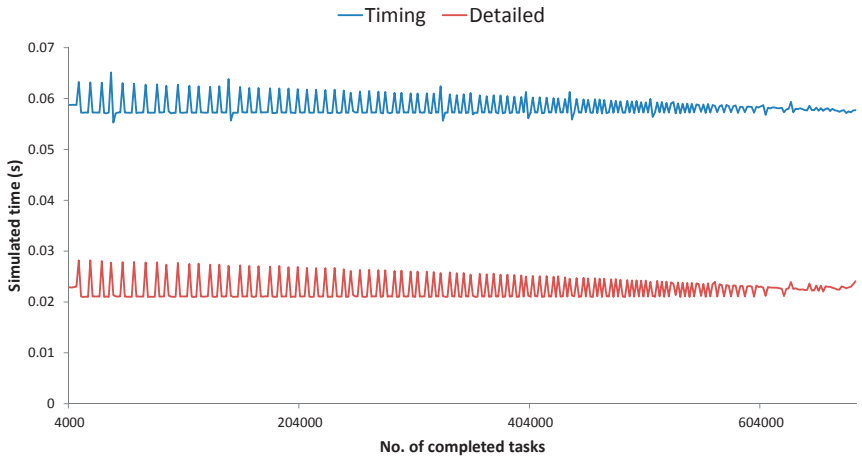


Figure C.8: SparseLU - Timing mode profile vs. Detailed mode profile

environment. To adapt to this environment, we propose the number of completed tasks as a progress metric for sampling. Our experiments over a large design space show that FASTA estimations error can be below 4% for a certain class of appplications. At the same time FASTA simulations can achieve up to 12x speedup. Applications that have a deep task tree, show increasing estimation errors with the core count. The reason for these errors is the interleaving of threads that causes a different task stream for each simulations mode. As a result, the representative samples calculated from a low-detail simulation profile are not representative for the detailed simulation. We also determined that the level in which the task stream changes between runs is correlated with the number of parent-nodes in an application's task tree. Our parameter analysis shows that for $K = 6$ clusters, in most cases the estimation error of FASTA is below 5%. These results are encouraging and they recommend FASTA for future research.

Bibliography

[1] A. R. Alameldeen and D. A. Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17, July 2006. ISSN 0272-1732. doi: 10.1109/MM.2006.73. URL http://dx.doi.org/10.1109/MM.2006.73.

[2] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, M. D. Hill, D. A. Wood, and D. J. Sorin. Simulating a $2M Commercial Server on a $2K PC. *Computer*, 36(2), 2003.

[3] K. C. Barr, H. Pan, M. Zhang, and K. Asanovic. Accelerating Multiprocessor Simulation with a Memory Timestamp Record. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, pages 66–77, March 2005. doi: 10.1109/ISPASS.2005.1430560.

[4] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The GEM5 Simulator. *SIGARCH Comput. Archit. News*, 39, 2011.

[5] J. Chen, M. Annavaram, and M. Dubois. SlackSim: a platform for parallel simulations of CMPs on CMPs. *SIGARCH Comput. Archit. News*, 37, 2009.

[6] Cilk++. Cilk++: A quick, easy and reliable way to improve threaded performance. Available: http://software.intel.com/en-us/articles/intel-cilk-plus/. (retrieved November 2016).

[7] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing State Loss For Effective Trace Sampling of Superscalar Processors. In *Proc. of the International Conference on Computer Design, VLSI in Computers and Processors*, 1996.

[8] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in

OpenMP. In *Proceedings of the 2009 International Conference on Parallel Processing*, pages 124–131, Sept 2009. doi: 10.1109/ICPP.2009.64.

[9] K.-F. Faxén. Wool - A Work Stealing Library. *SIGARCH Comput. Archit. News*, 36(5):93–100, June 2009. ISSN 0163-5964. doi: 10.1145/1556444.1556457. URL http://doi.acm.org/10.1145/1556444.1556457.

[10] S. Fuller and L. Millett. Computing Performance: Game Over or Next Level? *Computer*, 44(1):31–38, Jan 2011. ISSN 0018-9162. doi: 10.1109/MC.2011.15.

[11] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A New SPEC Benchmark Workload for Simulation-Based Computer Architecture Research. *IEEE Comput. Archit. Lett.*, 1(1), 2002.

[12] D. Leijen, W. Schulte, and S. Burckhardt. The Design of a Task Parallel Library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640106. URL http://doi.acm.org/10.1145/1640089.1640106.

[13] MacQueen, J.B. Some Methods for Classification and Analysis of Multivariate Observations. In *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*, pages 281–297, Berkeley, California, 1967. University of California Press. URL http://projecteuclid.org/euclid.bsmsp/1200512992.

[14] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, Jan 2010. doi: 10.1109/HPCA.2010.5416635.

[15] E. Perelman, G. Hamerly, and B. Calder. Picking Statistically Valid and Early Simulation Points. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, pages 244–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2021-9. URL http://dl.acm.org/citation.cfm?id=942806.943854.

[16] A. Rico, F. Cabarcas, C. Villavieja, M. Pavlovic, A. Vega, Y. Etsion, A. Ramirez, and M. Valero. On the simulation of large-scale architectures using multiple application abstraction levels. *ACM Trans. Archit. Code Optim.*, 8(4), 2012.

[17] TBB. Intel Threading Building Blocks. http://software.intel.com/sites/products/documentation/hpc/tbb/getting_started.pdf. (retrieved November 2016).

[18] T. F. Wenisch, R. E. Wunderlich, B. Falsafi, and J. C. Hoe. Simulation sampling with live-points. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 2–12, March 2006. doi: 10.1109/ISPASS.2006.1620785.

[19] T. F. Wenisch, R. E. Wunderlich, M. Ferdman, A. Ailamaki, B. Falsafi, and J. C. Hoe. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro*, 26(4):18–31, July 2006. ISSN 0272-1732. doi: 10.1109/MM.2006.79.

[20] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. Statistical Sampling of Microarchitecture Simulation. *ACM Trans. Model. Comput. Simul.*, 16(3):197–224, July 2006. ISSN 1049-3301. doi: 10.1145/1147224.1147225. URL `http://doi.acm.org/10.1145/1147224.1147225`.

[21] J. J. Yi and D. J. Lilja. Simulation of Computer Architectures: Simulators, Benchmarks, Methodologies, and Recommendations. *IEEE Trans. Comput.*, 55(3), 2006.

[22] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and Comparing Prevailing Simulation Techniques. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, HPCA '05, pages 266–277, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2275-0. doi: 10.1109/HPCA.2005.8. URL `http://dx.doi.org/10.1109/HPCA.2005.8`.

Appendix D

Paper C.1

On the Energy Footprint of Task Based Parallel Applications

Alexandru C. Iordan, Magnus Jahre and Lasse Natvig

International Conference on High Performance Computing and Simulation (HPCS)
2013

Abstract

From HPC systems to embedded devices, energy consumption is becoming a dominant factor in managing costs. With Chip multiprocessors becoming the platform of choice in almost all ICT segments, software developers need to employ parallel programming to fully exploit this architecture. However, parallelization adds a management overhead to the execution of an application. In this paper, we study parallel applications implemented using two TBP libraries, Wool and Intel TBB. We explore both compute and memory bound executions. We also investigate the energy footprint of the parallelization overhead and the effect it has on the energy-efficiency of the executing system. Our study looks into the behavior of some basic parallelization operations like task spawning, task synchronization and task stealing. We encountered situations when the number of task stealing operations grows exponentially with the core count increasing execution time. This behavior drastically reduces the energy-efficiency of those executions. Avoiding such behavior is crucial if future parallel systems are to reach their performance potential. Our results also show that the energy efficiency for compute intensive applications improves with the core count while for memory intensive application that is not the case.

D.1 Introduction

With increasing concerns about energy cost and battery life, almost all ICT segments today focus on reducing power requirements and saving energy. For HPC systems and data centers, high energy consumption translates into high cost of operation. High performance CPUs dissipate a lot of heat and require powerful cooling systems to keep them running. This results in over 30% of the energy consumed for solving an HPC task being "wasted" on cooling [1]. For embedded and mobile systems, energy consumption relates to battery life and the amount of time the device can operate without recharging or changing its battery.

For high performance CPU architects, power density and not energy itself is the main development constraint. In order to mitigate this *power wall* along with other limitations like the *memory wall* or the *ILP wall*, CPU designers introduced Chip multiprocessors (CMPs). This architecture makes it possible to utilize the increasing transistor count provided by improvements in production technology without a power budget growth [8]. Exploiting both instruction and thread level parallelism, CMPs allow an increase in aggregate performance that was not possible through traditional superscalar approaches. However, CMP architectures can not be fully exploited using traditional sequential programming. Parallel programming is needed in order to take full advantage of their resources and ultimately improve the energy efficiency of the system. For this reason, there is a considerable interest in programming models that can simplify parallel programming and make it available for mainstream programmers. We focus on *Task Based Programming* (TBP), a parallel programming model that has received significant attention recently [3, 7, 11, 17].

This work is a first step towards understanding the energy overheads of TBP applications. To achieve this, we study the energy footprint of the overhead introduced by parallel programming. We define the *energy footprint* as the energy spent for executing the given application or section of code in the context of the test system. We first investigate the applications as a whole and then look at key parts of TBP parallelization like task spawning, task synchronization and task stealing/migration. We chose these task management operations because they are the fundamental building blocks of TBP libraries.

To allow for extensive, noninvasive measurements in our study, we use a performance simulator and a power estimation tool. We use micro-benchmarks to study the parallel overhead and its effect on both compute and memory bound TBP applications. We parallelized our benchmarks using the Wool [7] and TBB [17] libraries and we computed the energy footprint for the basic TBP operations mentioned above. Our results show that task stealing operations can become real "power hogs" as the core count increases. Such behavior needs to be avoided for TBP libraries to reach their performance potential on future processors.

Algorithm 3 Wool scheduling loop

```
procedure DO_WORK( )
    i = 0
    while True do
        if i > 0 then
            i = i -1
            victim_thread = victim_thread + 1
        else
            i = random()
            victim_thread = i
        next_task = steal (victim_thread)
```

D.2 Task Based Programming Background

TBP libraries allow programmers to exploit concurrency by partitioning an application into tasks. Task are small regions of code that perform a specific function and that can be executed in parallel with other tasks. Dependencies among tasks are either specified by the programmer [7] or are inferred automatically based on annotation of procedure declarations [5, 15]. At runtime, tasks are distributed automatically to available processing units and all dependencies are enforced. This approach frees the programmers from managing and mapping the parallel threads onto the CMP.

For TBP, there are two programming styles that can be used to define the control flow. *Direct style* includes a synchronization operation that can be used to wait for the completion of a set of tasks before starting another one. This approach gives the programmer the flexibility to alter the control flow but also the responsibility to ensure that all tasks are synchronized [7]. The second style is called *continuation-passing style* where the completion of the current tasks returns a pointer to the next task to be executed. This programming style creates portable code and exposes optimization opportunities for compilers by simplifying the control flow [17]. In our study, Wool is a direct style library while Intel's TBB has a continuation-passing approach.

D.2.1 The Wool Programming Model

Wool was developed by Karl-Filip Faxén at the Swedish Institute of Computer Science. The main goal of Wool is to provide a simple programming interface and to ensure a very low parallel overhead. It is built on top of pthreads for thread management and has a scheduler based on a work-stealing. Being a direct style library, Wool facilitates the parallelization of existing code [7].

The user can supply the number of threads an application will start through a command line argument. To ensure the best distribution of work, the recommendation is to use 1 thread per physical core. Each worker maintains a set of data structures for implementing task management, in particular a pool of tasks that are ready to execute. Tasks are created

Algorithm 4 TBB scheduling loop

```
procedure WAIT_FOR_ALL( )
    repeat
        repeat
            while task is available do
                next_task = current_task->execute()
                Decrease ref_count for parent of current_task
                if ref_count == 0 then
                    next_task = parent of current_task
            current_task = get_task()
        until current_task == NULL
        current_task = steal_task(random_victim())
        if steal unsuccessful then
            Wait a fixed amount of time
            if waited too many times then
                Suspend and wait for signal from master thread
    until root_task is dead
```

(spawned) and added to the task pool of the worker that spawn them. From the task pool, tasks can be executed by their owner or can be stolen by other workers. Load balancing is achieved through work-stealing. The user can control this mechanism by supplying the number of tasks that are marked as *stealable* in each pool. When an application is started, only the main thread has work to do and all other workers start an infinite loop trying to steal work (see Algorithm 3). Tasks can spawn other tasks, resulting in a hierarchical task tree where the execution of a task is dependent on the completion of all its "children".

D.2.2 The Intel TBB Programming Model

TBB is a C++ template library design by Intel to help programmers create portable, parallel applications using task parallelism. The library includes algorithms, highly concurrent containers, locks and atomic operations, a task scheduler and a scalable memory allocator. Like Wool, load balancing is achieved through work-stealing. TBB provides an increased parallelism abstraction that avoids the low level programming inherent in the direct use of threading packages such as pthreads [17].

When TBB is first initialized, a set of worker threads is started and the calling thread becomes the master thread. Like Wool, the user can supply the number of threads through a command line argument. Each worker is assigned a task queue which is used to enqueue parallel tasks. The scheduling loop that runs on each worker consists of 3 nested loops responsible to obtain work(see Algorithm 4).

D.3 Basic TBP functions

In this paper, we study the impact the parallelization overhead has on the execution performance and ultimately on the its energy-efficiency. To do that, we focus on some basic operations of TBP libraries like task spawning, task synchronization and task stealing. For each of them, we estimate an energy footprint in order to identify potential "energy hogs". In the next subsection, we summarize how both Wool and TBB implement these basic TBP operations.

D.3.1 Wool Macros

Wool provides a set of task definition macros in the form:

TASK_n{rtype, name, arg_type1, arg1, ... , arg_typen, argn}

where *n* defines the arity (number of arguments) of the task, *rtype* is the type of the return value, *name* is the name of the task and *arg_typei* is the type of the *i*th argument. For a task called *example_task* defined in the above manner, Wool provides the following operations [6]:

SPAWN{example_task, arg1, ... argn}

SYNC{example_task}

CALL{example_task, arg1, ... argn}

SPAWN makes the task available for execution by adding it to the thread's task pool. A task is designated *stealable* if it is among the first *s* tasks spawned, where *s* is a command line argument. Marking a tasks as *stealable* adds a small overhead to the spawning operation. From the task pool, tasks can be executed either by the worker that spawned them (*inlined* execution) or by another worker after a stealing operation. When executed, a task can spawn other tasks which will result in a task tree with hierarchical dependencies.

SYNC is used to invoke the most recent spawned task in the pool having the name given as argument. This operation can succeed if that task is in the task pool of the current worker or it can fail if the task was stolen by another worker. If the operation fails, the current thread will try to steal some work so that it keeps busy. However, in order to avoid a deadlock situation, it will only steal from the thief that stole the original task. This method is called *leapfrogging* [19].

CALL is an optimized operation for task invocation, being equivalent to a SPAWN followed by a SYNC. Since the calling thread is the one who will execute it, the tasks is neither added to task pool nor marked as *stealable*. This makes this function faster than a SPAWN succeeded by a SYNC.

Wool also provides macros for parallelizing loops. The *LOOP_BODY* family of macros is used to defined the body of the loop and the *FOR* macro is used to execute the iterations in parallel.

LOOP_BODY_n{name, grainsize, index_type, index,

arg_type1, arg1, ... , arg_typen, argn}

where *n* defines the arity, *name* is the name of the loop body, *grainsize* is used to balance parallelism versus overhead, *index* is the index variable of the loop and *index_type* is its type. *arg_typei* refers to the type of the *i*th argument.

A parallel loop is invoked using

FOR{name, b_low, b_high, arg1, ... , argn}

where *name* is the name of the loop body, *b_low* and *b_high* are the iteration bounds and *argi* are loop invariant arguments.

Apart from the main thread, all workers are executing a procedure called *Do_work()* (see Algorithm 3). This procedure is basically an infinite loop for stealing work. In order to reduce execution time, Wool implements a quasi-random selection of the victim thread: workers are scanned sequentially and only once in a while a randomize function is called. Since a task can spawn other tasks, a worker can populate its task pool after only one successful steal and will have to empty it before returning to the *Do_work()* procedure.

In our experiments we estimate the energy-footprint of the SPAWN and SYNC operations as well as the *steal()* function. We also investigate the reasons for the different outcomes of all these operations.

D.3.2 TBB Methods

In TBB applications, tasks are described as C++ classes derived from the *tbb::task* base class. The *tbb::task* class includes a virtual method called *execute()* used for describing the body of the task. The programmer needs to define this method for each task class that he creates.

Once a task has been instantiated, it can be launched by a *spawn(task)* method and added to the task queue. From the task queue the task is available for execution or stealing. A task can instantiate and spawn other tasks creating a hierarchical task tree.

TBB allows parallelism to be annotated both explicitly, through *spawn(task)*, and implicitly , through some templates. A list of these templates is given below:

parallel_for <*range, body*>

parallel_reduce <*range, body*>

parallel_scan <*range, body*>

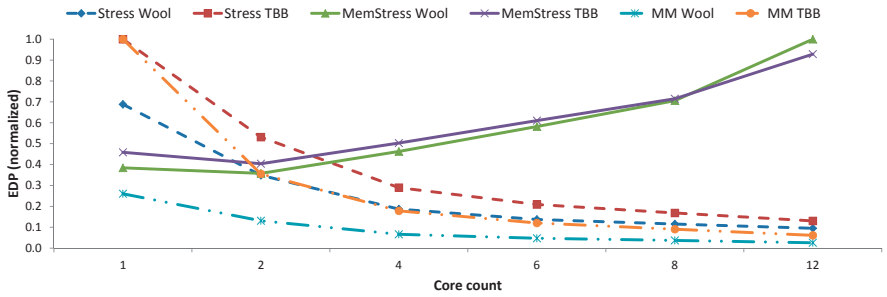


Figure D.1: Normalized EDP of the three micro-benchmarks for both Wool and TBB libraries

parallel_while <*body*>

parallel_sort <*iterator, compare*>

Each worker thread is running a scheduling procedure called *wait_for_all()* consisting of 3 nested loops which is shown in Algorithm 4. The inner loop is executing the current task by calling the *execute()* method. Since TBB is a continuation-passing style library, the completion of this tasks returns a pointer to the next task that needs to be executed. If a new task is not referenced, the inner loop exits. In the middle loop the *get_task()* method tries to dequeue the local task queue using a LIFO order. If successful, the inner loop is called again. If unsuccessful because the queue is empty, the middle loop exits and the outer loop invokes the stealing mechanism. If stealing fails, the current worker backs off for a period of time and then selects randomly a new victim. While fast and easy to implement, this method is not fair: the same worker can be selected as victim several times even if it does not have the largest task queue [4].

In this work, we investigate the energy-footprint of the *spawn()*, *get_task()* and *steal()* methods.

D.4 Methodology

D.4.1 Simulation tools

We performed our experiments using a new parallel x86 computer architecture simulator called Sniper [2]. Sniper uses the interval core model [9] and Graphite simulation infrastructure [14] to provide fast and accurate simulations. We modeled a Nehalem-based Xeon 5500-series multi-core CPU (code name Gainestown) with a clock frequency of

Table D.1 Main characteristics of modeled processor

	Parameter	Value		
Core	Clock frequency	2.66 GHz		
	Instruction set	x86-64		
	Dispatch width	4		
	Window size	128		
	Core count	1-,2-,4-,6-,8-,12-cores		
Cache		Size	Assoc.	Replacement policy
	L1 iCache/dCache	#cores x 32KB	4/8	LRU
	L2 Cache	#cores x 256KB	8	LRU
	L3 Cache	2/4/8/12/16/24 MB	16	LRU
Main memory	Size	2/4/8/12/16/24 GB		
	Number of Memory Controllers	1/2/4/6/8/12		

2.66 GHz and 3 levels of cache. Table F.1 presents the main characteristics of the modeled processor.

The performance results from Sniper are fed into a power estimation tool called McPAT [13]. An important characteristic of McPAT is its ability to model dynamic, static and short-circuit power. Dynamic power refers to the power required by a circuit to switch from one logical state to the other. For each system component, dynamic power is defined as: $power_{dynamic} \sim AF \cdot C \cdot V_{dd}^2 \cdot f$, where *AF* is the activity factor, *C* is the total load capacitance, V_{dd} is the supply voltage and f is the clock frequency [10]. Switching circuits also dissipate short-circuit power which McPAT modeles analytically. Static power is caused by current leakage during periods of non-activity. McPAT estimates leakage current using models of real-world CMOS circuits.

All our benchmarks were parallelized using Wool version 0.2 and TBB version 4.1.1. In order to isolate and measure only the overheads introduced by task spawning, task synchronization and task stealing we customized both libraries. We added some special instructions called *markers* in the beginning and at the end of each function of interest. These markers are recognized by Sniper and a label is added to results reported when executing the region of code in between two markers.

D.4.2 Benchmarks

In this work, we use micro-benchmarks to measure the parallelization overhead with as little interference as possible from other sections of the execution. *Stress* is a recursive micro-benchmark that creates a balanced binary task tree. The leaf nodes perform a number of additions while most of the execution is comprised from spawning the task tree and load-balancing the workload through task stealing. The user can change the number of operations performed by the leaf nodes as well as the depth of the task tree through com-

mand line arguments. This is a compute-intensive benchmark having no memory traffic outside the on-chip caches.

To test and measure the impact of the parallelization overhead on a memory-intensive execution we used another micro-benchmark called *MemStress*. This applications creates the same balanced task tree as *Stress*. The difference is that the leaf nodes of *MemStress* perform simple operations with the elements of a large array. The array is large enough that it can not be stored in the private caches and its elements are accessed in a non-sequential way. Again, the user can control the size of the array, the depth of the task tree as well as the stride used to read the array's elements.

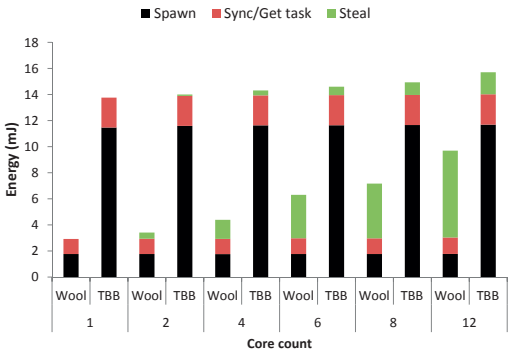
The third micro-benchmark we use is called *MatrixMul*(MM) and performs a matrix multiplication. The result of the multiplication is computed using subarrays (or blocks) of the operands rather than the whole arrays. With this approach, the small blocks can be loaded into the private caches allowing for a faster computation. By modifying the size of the blocks, this application can be either compute- or memory-intensive. Another particularity of this micro-benchmark is that it uses parallel FOR operations to express parallelism compared to the explicit task creation of *Stress* and *MemStress*.

D.5 Results

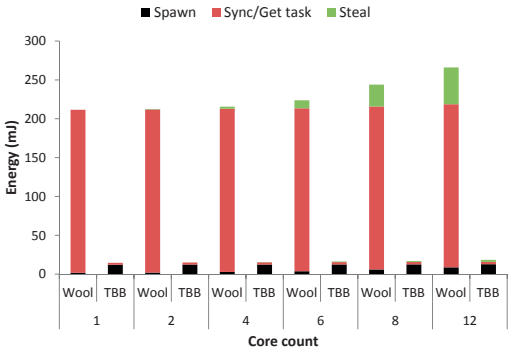
Before investigating the task management operations individually, we first take a look at the parallel executions as a whole. To quantify the energy-efficiency of each execution we use the *Energy-Delay Product* (EDP) metric. EDP ensures a balanced comparison among the test systems by putting equal weight on both energy consumption and performance. To allow a better visualization of all results in Fig. D.1 we normalized our results. For each benchmark, we normalized to the maximum value of each pair of executions (Wool and TBB) across all core counts. Consequently, it is unfair to compare the values of different benchmarks in Fig. D.1. *Stress* (dashed lines) and *MM* (dashed and dotted lines) show the typical trend of a compute-intensive application: execution time (and consequently EDP) decreases with the core count. *MemStress* shows the exactly opposite trend: more cores translate into more conflicts in accessing the memory which leads to longer execution time (and higher EDP values). The high concurrency for hardware resources has also an effect on the amount of parallel overhead in Wool. In Fig. D.1, the Wool version of *MemStress* overtakes the TBB version for the 12-core platform. This happens because the overhead of stealing grows with the core count which is discussed later in this section.

There are two challenges that need to be addressed when parallelizing an application: ensuring a balanced workload and reducing the parallel overhead. Both Wool and TBB use work-stealing to distribute tasks among the worker threads and both mechanisms achieve a good balance. If there are enough tasks to be executed, both libraries will distribute them among available threads which ensures a good use of hardware resources.

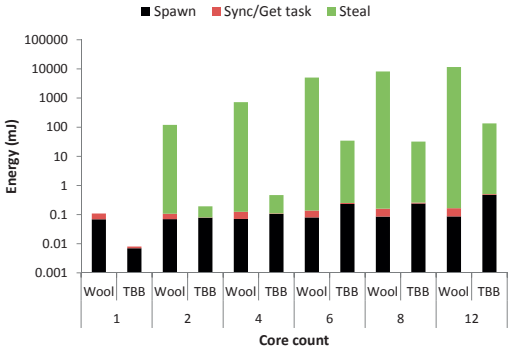
The functions/methods we studied in this paper are all part of the parallelization overhead.



(a) Stress



(b) MemStress



(c) MM

Figure D.2: Energy footprint

The number of times these functions are called as well as the way they manage data impact the performance of the application and ultimately the energy-efficiency of the system. We estimate the power requirements of the system when executing these functions and computed the energy footprint of each of them.

Fig. D.2 and D.3 show the results of our energy footprint study. Fig. D.2 shows the energy footprint of the task management operations while Fig. D.3 shows how much of the total execution's energy all the task management operations account for (in percentage). For a better visualization of MM's results in both Fig. D.2 and D.3, we used a logarithmic scale.

In Fig. D.2 is apparent that for both libraries, the energy footprint for task spawning and synchronization does not increase with core count. This happens because the number of spawns and syncs is related to the size of the input set which we kept constant across all simulation platforms.

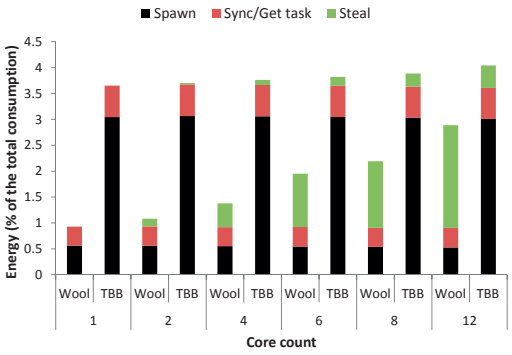
The stealing operations on the other hand show an increase of the energy footprint with the core count. This is due to the fact that stealing operations can be unsuccessful. A failed attempt forces the thief (the thread that is trying to steal) into a spinning cycle before it can try to steal again. Attempting to steal work from a thread with an empty queue or race conditions between two thieves are the most common situations for an unsuccessful steal for Wool. Stealing can also fail if the victim thread has no tasks marked as stealable. Wool has a very aggressive stealing mechanism which translates into a large number of unsuccessful attempts.

In Wool, each worker thread tries to execute the tasks in its pool in LIFO order. It can happen that a synchronization operation may fail because the task was stolen by another thread. In this situation, the current thread will try to steal work from the thief thread in order to keep busy (the *leapfrogging* technique). This means that a thread may be forced to steal tasks even if his own pool is not empty.

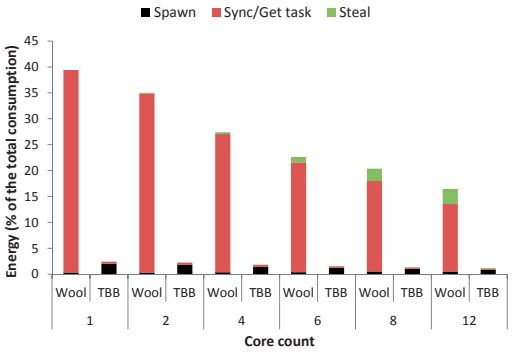
As the number of races for hardware resources increases with the core count, so is the number of unsuccessful steals. This means that stealing operations take up a larger part of the execution as the number of cores increases (see Fig. D.2). This can be seen in Fig. D.1 also. There, the high number of unsuccessful steals in the Wool version of *MemStress* gives it a higher EDP value than the TBB version for the 12-core platform.

TBB has a different approach for task stealing and execution. As mentioned before, the scheduling loop that runs on each worker thread consists of 3 nested loops (see Algorithm 4). In addition, after several unsuccessful attempts, a thief will suspend execution and wait for the main thread to generate more work. As a result, TBB applications perform far less stealing operations than Wool ones. Also, the number of TBB steals does not increase with the core count as fast as the Wool ones. This means that for high number of cores, stealing will not be the dominant task management operation in the execution (see Fig. D.2).

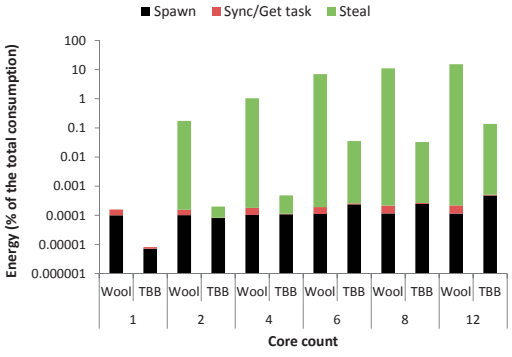
Fig. D.2 also shows a big difference between the energy footprint of Wool's sync operation



(a) Stress



(b) MemStress



(c) MM

Figure D.3: Energy footprint - percentage of the total execution

for the three benchmarks. Because of stealing, there can be more than one worker who tries to acquire a task. To ensure a "first come first served" order during races, the sync operation includes a memory fence. This fence forces the current worker thread to wait for the completion of all memory operation issued up to the current time before it can try to acquire the task. For Stress and MM there are few such memory operations so the syncs are fast. However, for MemStress many memory accesses are issued and the sync operations spend a lot of time waiting for their completion. In this regard, there is a big potential for energy-efficiency improvements in Wool.

Finally, the results of MM give us an insight into the role the tasks granularity plays in the parallelization overhead. MM creates a relative small number of tasks compared to the other two benchmarks (899 vs 32767 tasks) which means there are more races in acquiring the tasks. Again, the aggressive stealing makes Wool "waste" a lot of energy on failed attempts. Stealing operations become the dominant starting with the 2-core system, when 120 mJ are consumed for stealing, 70 μJ for spawning and 37 μJ for synchronization. Spawns and syncs consumption does not increase much with the core count but stealing ends up consuming 11.5 J for the 12-core system, 15.3% out of the total consumption (see Fig. D.3). By contrast TBB consumes 473 μJ for spawning, 32 μJ for the *Get task()* method and 134.2 mJ for stealing for the 12-core system. These results show that TBB is better suited to deal with coarse task granularity than Wool.

In Fig. D.3, both Stress and MM show the same trend as in Fig. D.2. The overhead of stealing grows with the core count (for both libraries) and in Wool's case it becomes the dominant component.

Because MemStress is a memory bound application, increasing the number of cores does not lead to a reduction of execution time. When you couple this with the fact that power requirement increases with core count you end up with a trend for energy consumption similar with the EDP trend in Fig. D.1. The trend of MemStress in Fig. D.3 can be explained by the fact that the energy footprint of the task management operations does not increase that much with core count (see Fig. D.2) while the total energy consumption of the application does.

D.6 Related work

To the best of our knowledge, there is no other study that quantifies the energy consumption of parallel overhead for any TBP libraries. Instead, many authors have focused on characterizing and improving existing parallel libraries from the performance point of view or doing comparative studies of several libraries.

Vandierendonck et al. advocate the use of TBP models with nested task spawning for writing general-purpose programs [18]. The authors developed a Cilk-like language to express parallel pipelines and extended a Cilk-like scheduler to recognize and enforce argument dependency types on task spawns. This programming model enhances the ease

of programming parallel pipelines by expressing the parallelism densely.

Contreras and Martonosi study and characterize some of the overheads of Intel's TBB [4]. They concluded that task management operation can have a detrimental effect on the performance of parallel execution for core counts higher than 8 cores. The authors also note that random stealing fails to scale with increasing core counts and that alternative policies that consider the current state of the runtime library can improve performance in these situations.

Podobas et al. do a performance comparative study of several TBP libraries, including Wool [16]. They use both micro-benchmarks and a subset of the BOTS suite to characterize application performance and the costs for task creation and stealing. The study concludes that Wool has the lowest overhead for task spawning and task stealing.

Li and Martinez studied the power-performance implications of running parallel applications on CMPs [12]. Using both an analytical model and detailed simulations, the authors show that parallel computing can bring significant power savings and still meet a given performance target. These savings can be achieved through judiciously selections of the granularity and voltage/frequency levels which illustrates the dependency of the optimum operating point on multiple interacting factors.

D.7 Conclusions

Due to limitations like the power wall, the memory wall and the ILP wall, CPU development could not longer follow the performance increase trend the superscalar architecture enjoyed. The CMP architecture was introduced to mitigate these development constraints and to provide a performance increases for new generations of CPUs. This new hardware architecture can not be fully exploited with traditional sequential programming and parallelization needs to be employed. To ensure the best utilization of the hardware resources and an increased energy-efficiency of the system, parallelization overhead needs to be addressed. In this paper, we investigated the energy footprint of some basic parallelization operations for two parallel libraries. We used micro-benchmarks to isolate and measure the parallelization operation for both compute and memory intensive executions.

Our results show that Wool parallelized benchmarks have a smaller energy footprint compared to the TBB versions. This is not surprising, since one of Wool's development goals was to have a small parallel overhead. However, the parallelization approach used in Wool is far from being the most energy-efficient. The stealing mechanism is very aggressive and the number of unsuccessful steals grows exponentially with the core count. This approach makes the energy cost of stealing operations increase with the core count. In contrast, TBB has a far more "peaceful" implementation for task stealing. Threads that fail to acquire work several times enter a sleep state and are awoken by the main thread when work becomes available. This translates into significantly less failed steals compared to Wool executions. However, TBB is a much more complex library and includes

more constructs to help the programmer express parallelism in more ways. It also has a higher abstraction level to hide this complexity from the programmer. All these require more "management operations" (many others than those studied in this paper) that translates into a longer execution times.

In regard to other parallelization functions, our results show that both libraries handle well operations like task spawning and task synchronization. There is no indication that these task management operations do not scale well with the core count for either libraries. With better care for the data structures used in *Spawn()* and *Get_task()* methods, the results of TBB can be significantly improved.

With this research we want to promote a more energy aware parallel programming. We think that by designing parallel software that can better exploit current multi-core chips, we can address an issue that has increased in importance over the last years in ICT field: energy consumption. Developing parallelization libraries that can scale with core counts and that have a dynamic approach to solving workload imbalance is a first step towards that goal. TBB and Wool both qualify in this respect, but both of them can be improved from an energy-efficiency point of view. We think this topic is promising and will continue our study in future work.

Bibliography

[1] L. A. Barroso and U. Höelzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[2] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, Nov 2011.

[3] Cilk++. Cilk++: A quick, easy and reliable way to improve threaded performance. Available: `http://software.intel.com/en-us/articles/intel-cilk-plus/`. (retrieved November 2016).

[4] G. Contreras and M. Martonosi. Characterizing and Improving the Performance of Intel Threading Building Blocks. In *IEEE International Symposium on Workload Characterization*, pages 57–66, Sept 2008. doi: 10.1109/IISWC.2008.4636091.

[5] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas. OmpSs: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters*, 21, 2011.

[6] K.-F. Faxén. Wool 0.1 users guide. `http://www.sics.se/~kff/wool/users-guide.pdf`.

[7] K.-F. Faxén. Wool - A Work Stealing Library. *SIGARCH Comput. Archit. News*,

36(5):93–100, June 2009. ISSN 0163-5964. doi: 10.1145/1556444.1556457. URL http://doi.acm.org/10.1145/1556444.1556457.

[8] S. Fuller and L. Millett. Computing Performance: Game Over or Next Level? *Computer*, 44(1):31–38, Jan 2011. ISSN 0018-9162. doi: 10.1109/MC.2011.15.

[9] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval Simulation: Raising the Level of Abstraction in Architectural Simulation. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 307–318, Feb. 2010.

[10] S. Kaxiras and M. Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan and Claypool Publishers, 2008.

[11] D. Leijen, W. Schulte, and S. Burckhardt. The Design of a Task Parallel Library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640106. URL http://doi.acm.org/10.1145/1640089.1640106.

[12] J. Li and J. Martínez. Power-Performance Considerations of Parallel Computing on Chip Multiprocessors. *ACM Trans. Archit. Code Optim.*, 2, 2005.

[13] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multi-Core and Many-Core Architectures. In *International Symposium on Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM*, pages 469–480, Dec 2009.

[14] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, Jan 2010. doi: 10.1109/HPCA.2010.5416635.

[15] J. Perez, R. Badia, and J. Labarta. A Dependency-Aware Task-Based Programming Environment for Multi-Core Architectures. In *IEEE International Conference on Cluster Computing*, 2008.

[16] A. Podobas, M. Brorsson, and K.-F. Faxén. A Comparison of Some Recent Task-based Parallel Programming Models. In *Third Workshop on Programmability Issues for Multi-Core Computers*, 2009.

[17] TBB. Intel Corporation. *Intel Threading Building Blocks Reference Manual*. Available: http://threadingbuildingblocks.org/. (retrieved November 2016).

[18] H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos. Parallel Programming of General-Purpose Programs Using Task-Based Programming Models. In *Proc. of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, 2011.

[19] D. B. Wagner and B. G. Calder. Leapfrogging: a portable technique for implementing efficient futures. In *Proc. of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1993.

Appendix E

Paper C.2

Victim Selection Policies for Intel TBB: Overheads and Energy Footprint

Alexandru C. Iordan, Magnus Jahre and Lasse Natvig

International Conference on Architecture of Computing Systems (ARCS)
2014

Abstract

With the wide adoption of Chip Multiprocessors (CMPs), software developers need to switch to parallel programming to reach the performance potential of CMPs and maximize their energy efficiency. Management overheads due to parallelization can cause sub-linear speedups and increase the energy consumption of parallel programs. In this paper, we investigate the parallelization overheads of Intel TBB with a particular focus on its victim selection policy. We implement an "all knowing" oracle victim selection scheme as well as a pseudo-random scheme and compare them against TBB's default random selection policy. We also break down TBB's parallelization overheads and report how basic operations like task spawning, task stealing and task de-queuing impact the energy footprint. Our experiments show that failed task stealing is by far the highest energy consumer. In fact, the oracle victim selection policy can reduce the application energy footprint by 13.6% compared to TBB's default policy.

E.1 Introduction

Energy consumption has become the main challenge for almost all systems in the information world, from HPC to embedded devices. Architects and developers are trying to find solutions for problems ranging from reducing the high cost of operation of data centers to maximizing the battery life of mobile and embedded systems. For over 20 years, techniques like transistor-speed scaling, pipelining, out-of-order execution and speculation have increased CPU performance at a rate of 50% per year [3]. However, diminishing returns from transistor scaling and power budget limitations has almost removed the single-core performance improvement trend.

The introduction of Chip Multiprocessors (CMPs) enabled the mitigation of development constraints like the *power wall* and the *ILP wall*. CMPs allow chip designers to utilize the increasing transistor count available with each new generation without increasing the power budget [10]. However, to fully take advantage of this architecture, parallel software is required since the performance potential of CMPs lies in exploiting thread level parallelism. This places a new burden on the software developers because there is no widely adopted programming model that facilitates easy parallelization. In this work, we focus on *Task Based Programming* (TBP) which is a parallel programming model that has received significant attention recently [6, 8, 13, 21].

To reduce the impact of parallelization overheads, a necessary first step is to identify the root cause of such overheads. To this end, we investigate the extra instructions added by parallelization management and the energy consumption of these instructions which we refer to as the *energy footprint*. More precisely, the energy footprint is the energy spent for executing the given application or section of code in the context of the test system.

In our experiments, we utilize Intel's Thread Building Blocks (TBB) [21] library for parallelization. TBB is a C++ template library designed to help programmers create portable, parallel applications using task parallelism. It was designed to avoid the low level programming inherent in the direct use of threading packages such as pthreads [21].

To allow for extensive and noninvasive measurements, we use a performance simulator and a power estimation tool in our study. We implement two victim selection policies in addition to TBB's random policy and report the performance and energy overheads of 5 PARSEC benchmarks [2]. We also break down TBB's overheads and look into basic TBP operations like task spawning, task stealing and task de-queuing. In our results, failed steals are the highest contributor to the overheads' energy footprint. With more accurate victim selection, the energy footprint of the application can be reduced by up to 13.6%.

E.2 Intel TBB

The concept of parallel programming is almost as old as the computer itself. Over the years, many parallel languages have been developed and a multitude of research was done

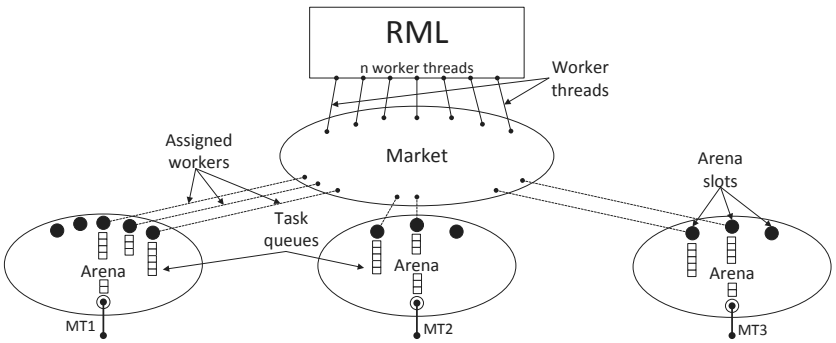


Figure E.1: Components of TBB's task scheduler

in an effort to improve raw performance and maximize hardware utilization [19]. With the majority of those approaches, one factor was often overlooked: the composability of the resulting solution. Composability of an applications refers to its ability to run efficiently side by side with other applications and being able to cope with the fact that it does not have exclusive access to the hardware resources [18]. In today's *multi-core era*, if parallel applications are not developed to dynamically scale and take advantage of all the resources that are available to them, the overall efficiency of the system suffers. In this work we focus on Intel's TBB version 4.1.1., which was design to ensure a high degree of composability.

TBB allows parallelism to be annotated both explicitly, by calling the *spawn()* method, and implicitly, through some templates like *parallel_for* or *parallel_reduce*. Tasks get created by the *spawn()* method and then added to the calling thread's task queue inside the arena (see Fig. F.1). From the arena the task is available for execution by its owner thread or by other workers through stealing. A task can instantiate and spawn other tasks resulting a hierarchical task tree.

When an application thread instantiates the *tbb::task_scheduler_init* object, that thread becomes a TBB master thread (MT). All threads created by TBB to help complete the work of the MT are called *worker threads*. The *Resource Management Layer* (RML) is the component that hosts the pool of worker threads and gets instantiated first (see Fig. F.1). No worker threads are created at this point, this being postponed until the first task is spawned.

Next a *Market* component is instantiated. This component was added in version 3.0 of TBB to ensure the composability of the framework. It guarantees that the work (the tasks) of one MT are isolated from other MTs that may be executing on the same machine. The role of the market is to assign workers to the arenas of each MT. The limit of the total number of workers available is set to 1 less than the maximum of the argument of the *tbb::task_scheduler_init* constructor and the total number of logical CPUs on the

executing system.

Finally, the *Arena* associated with calling MT gets allocated. An arena encapsulates all the tasks and the execution resources (worker threads) available to a MT. Each arena is assigned a number of slots representing the number of workers the arena requires to perform its parallel tasks. This is defined as 1 less than the minimum of the argument of the *tbb::task_scheduler_init* constructor and the total number of workers available (limit set by the market). Because several MTs can coexist, the total number of workers requested by all arenas can be greater than the number of workers available in the RML's pool. In this situation, the market will allot workers proportionally to each MT's request.

All these components and limits are created only once, during the first instance of the *tbb::task_scheduler_init* object in the current execution. If an MT is not the first one to call the task scheduler, it will only create a new arena that will comply with the limitation imposed by the market. Upon creation or destruction of an arena, the worker threads can migrate between the active arenas.

After they are created, each worker thread runs a scheduling procedure called *wait_for_all()* consisting of 3 nested loops. The inner loop is executing the current task by calling its *execute()* method. TBB is a continuation-passing style library which means that the completion of this task returns a pointer to the next task that needs to be executed. If a new task is not referenced, the inner loop exits. In the middle loop the *get_task()* method tries to dequeue the local task queue using a LIFO order. If successful, the inner loop is called again. If unsuccessful because the queue is empty, the middle loop exits and the outer loop invokes the stealing mechanism by calling the *receive_or_steal_task()* method.

E.3 The stealing mechanism

E.3.1 The TBB implementation

Stealing is part of the *receive_or_steal_task()* method. This method includes some other techniques to find a new task to execute than just stealing: mailing tasks via task-to-thread affinity mechanism, reload offloaded non-priority tasks, reload tasks abandoned by other workers. *Receive_or_steal_task()* method runs an infinite loop and calls each of the above mentioned mechanisms, stealing being the last one. Before a steal is attempted, a victim thread is selected randomly from the current arena. If the attempt is successful, the method returns and the scheduler re-enters the inner loop. If unsuccessful, a failure counter is incremented and the execution pauses before looping back. Also, if the failure counter surpasses a given threshold (default value is 100), the current worker thread is freed and returns to the RML.

The first step when a steal is performed is to use the *lock_task_pool()* method and try to get a lock on the victim. If the *lock_task_pool()* fails, the worker thread goes through a 5 steps exponential backoff. After 5 fails, the current thread yields its resources and

waits for its next time slot to try again locking the same victim. This locking mechanism assures the high composability of TBB we discussed in Section E.2. However, since we simulate 1 thread / hardware core, the yielding function returns immediately and the thief thread will continue to try to lock its victim.

A stealing attempt can fail for several reasons. The most common situation is selecting from a victim with an empty task queue. Applications with an unbalanced workload distribution face this problem often.

Race contention is also a common situation for failure. When 2 or more threads are trying to get exclusive access to the same task queue by calling the *lock_task_pool()*, only one can succeed. A thief can return from the *lock_task_pool()* only if it either succeeds or the victim's task queue has been depleted.

A special situation is when a thief thread is competing for access with the owner thread of that task queue. If there are more than 1 task in the queue, there is no race contention because the thief will steal at one end while the owner will dequeue the other. However, if there is only 1 task in the queue, the owner thread will have priority and the thief will backoff even if it already acquired the lock.

E.3.2 The oracle selection scheme

In an attempt to set an upper bound for the performance gain, we first implemented an "all knowing" scheme we call oracle selection. This method leverages on the fact that we use a simulator and not a real machine and it provides TBB with information that would be otherwise very "expensive" to obtain. We created a data structure to store the occupancy of all tasks queues in the arena as well as the level of congestion for each queue (the number of workers trying to steal from this queue). This structure is stored outside the simulated memory space in our simulator and is updated by the application through specialized instructions called markers. Since we do all this computation outside the simulated environment, our TBB application sees the victim selection as an extremely fast procedure. The queue with available tasks for stealing and with the lowest congestion level is selected as victim. This oracle scheme provides very fast and accurate results, but it is not optimal. There are still situations when updates to our structure do not propagate fast enough and the selected victim ends up creating conflicts.

E.3.3 The pseudo-random selection scheme

The second selection method we implemented is a pseudo-random scheme inspired by the Wool library [8]. For the first stealing attempt, we randomly select a task queue. If stealing from this victim fails, we then start a loop and sequentially scan the other active task queues, excluding the one of the current thread. In this way we will first try to steal from all possible queues before looping back in the *receive_or_steal_task()* and selecting a new random victim. Also, we removed the call to the yielding function

Table E.1 Main characteristics of modeled processor

Core		Cache			Main mem.	
#cores	1-,2-,4-, 8-,16-cores		Size	Assoc.		
Clock frequency	2.66 GHz	L1 i/dCache	#cores x 32KB	4/8	Size	2/4/8/ 16/32 GB
Instruction set	x86-64	L2 Cache	#cores x 256KB	8		
Dispatch width	4	L3 Cache	2/4/8/ 16/32 MB	16		
Window size	128					

from the *lock_task_pool()* and forced the method to return after the 5 steps exponential backoff. This approach eliminates the conflicts caused by the immediate return of the yielding function, but it will also make the stealing mechanism a bit more aggressive since it allows it to select new victims faster. It is worth mentioning that by doing this, we did not eliminate TBB's composability feature since yielding is implemented in more than one place.

E.4 Methodology

E.4.1 Simulation tools

We performed our experiments using a parallel, x86 computer architecture simulator called Sniper [4]. Sniper uses the interval core model [11] and Graphite simulation infrastructure [17] to provide fast and accurate simulations. Our model is based on a Nehalem-based Xeon 5500-series multi-core CPU (code name Gainestown) with a clock frequency of 2.66 GHz and 3 levels of cache. Table F.1 lists the main characteristics of the modeled processor.

The performance results from Sniper are fed into a power estimation tool called McPAT [15]. An important characteristic of McPAT is its ability to model dynamic, static and short-circuit power. Because we use only one CPU model, for a given core count the static power is a constant value. This is why for all our experiments we computed the energy footprint using only the dynamic power.

E.4.2 Benchmarks

For our experiments, we used the default TBB implementations of *Blackscholes*, *Bodytrack*, *Fluidanimate*, *Streamcluster* and *Swaptions* benchmarks with the medium input set

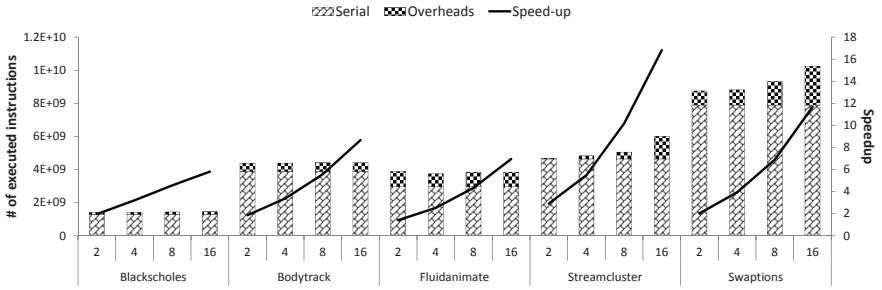


Figure E.2: Executed number of instructions and speedup

from the PARSEC suite [2]. Collectively, these benchmarks express parallelism both explicitly as well as through some templates like *parallel_for*, *parallel_reduce* and pipelines. They also employ some special TBB constructs like cache affinity partitioners and cache allocators. All these provide a wide test base for our study.

Parallelization was done using TBB version 4.1.1. which we customized in order to isolate and measure the overheads introduced by task spawning, task de-queuing and task stealing. We added special instructions called *markers* in the beginning and at the end of each function of interest to allow us to make measurements on the enclosed region of code.

To ensure statistically stable results, we performed 10 simulations of each benchmark for every core count. We averaged the performance results before estimating the power requirements. We computed the standard deviation (σ) of the execution time for each 10 simulation set as a percentage out of the average value for the set. Our results show a σ that ranges between 0.09% and 14.1% with no outliers (an outlier is a value that is above or below $3\sigma\pm$average value).

E.5 Results

Parallelization overheads often account for the sub-linear speedups of parallel implementation. While this still means that the work gets done faster, the energy required to complete the parallel execution is often equal or greater than the serial one. In Section E.5.1 we quantify these overheads as the difference in number of executed instructions between parallel and serial executions. We also break down the the overheads and see how task spawning, task de-queuing and task stealing impact the parallel execution and its energy footprint. For better visualization Fig. E.3 is plotted with logarithmic scale on the vertical axis. Finally, in Section E.5.2 we look into what performance and energy efficiency gains we can achieve by modifying the victim selection policy.

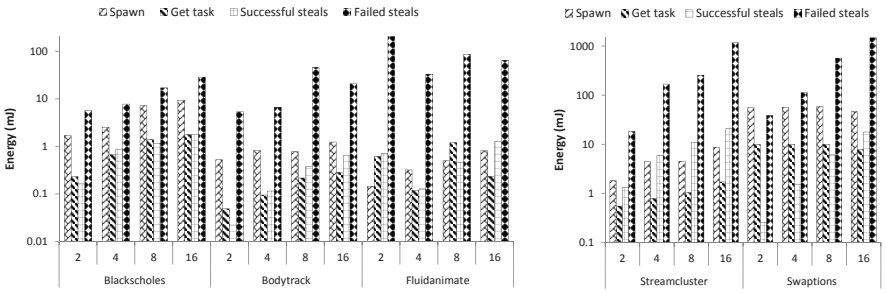


Figure E.3: Energy footprint for task management operations

E.5.1 Parallel overheads

Blackscholes employs TBB's *parallel_for* template for all the options in the input portfolio. Tasks are created by dividing these options on to the thread workers. This benchmark uses an *auto partitioning* algorithm to control the granularity of the tasks in order to handle work imbalance as well as possible. In Fig. E.2 you can see that the overheads are almost constant across the core count. This shows how small the parallel section is compared to the serial one and also why we see only a 5.8 speedup on the 16 core execution. Fig. E.3 breaks down the overheads and shows the energy footprint of the task spawning, task de-queuing and task stealing. Because we kept the input set constant across all core counts, our total number of tasks increases as we scale up the number of execution threads, but tasks also become finer. This has two consequences: the energy footprint for spawning increases from 2 to 16 cores (see Fig. E.3) and the overhead to useful work ratio per task increases with the core count. Fig. E.3 shows the same trend for the *get_task()* method. Second, we have the high number of stealing attempts. The energy footprint for failed steals is highest among what we measured and the trend is: more cores means more conflicts which leads to more failed attempts. Successful stealing has a smaller footprint but the same trend.

Bodytrack uses a 2 stage TBB pipeline construct to process the input images. In each stage *parallel_for* templates are used to divide the workload into parallel tasks. The difference compared to Blackscholes is that a special parameter of the *parallel_for* template, the *grain size*, is used to ensure a minimum size for each task. Similar with Blackscholes, Fig. E.2 shows a low parallel/sequential ratio as well as a sub-linear speedup. Because Bodytrack has larger sequential regions throughout the execution, the average number of threads that are active during the execution is lower than for the other benchmarks. This means that the worker threads return to RML (see Fig. F.1) because of work starvation more times. However, before returning, they attempt to steal 100 times each and fail which drives up the energy footprint (see Fig. E.3).

Fluidanimate computes the interactions between the particles of an incompressible fluid.

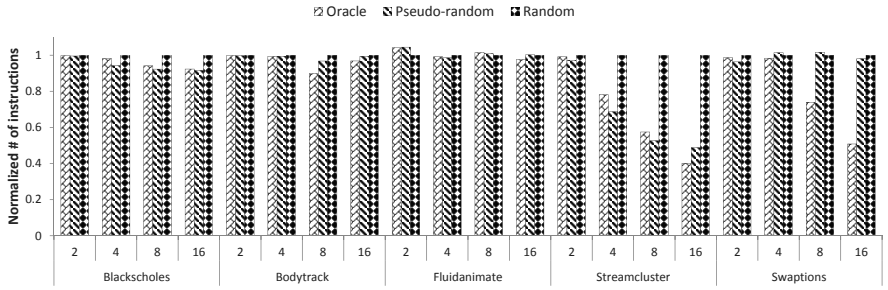


Figure E.4: Victim selection policies - comparison of overheads

Its input set, a matrix describing the positions of the particles, is divided into a grid of size N*M = the number of threads. For 2 threads we have a 1*2 grid, for 4 threads we have a 2*2 grid, for 8 threads we have 2*4 grid and for 16 threads we have a 4*4 grid. For each particle in the grid, the interactions with all its neighbors on the 8 surrounding directions are computed. When parallelized, this translates into larger lists of tasks that can be spawned for a square grid compared to a rectangular one. For this reason, our 4- and 16-cores simulations show fewer calls to the *get_task()* method and considerable less attempts to steal when compared to the 2- and 8-cores respectively (see Fig. E.3).

The Streamcluster results are the best in terms of speedup when comparing the parallel version to the serial one (see Fig E.2). This happens because the input set does not fit into the cache hierarchy and there is a lot of access to main memory when executed sequentially. The parallel tasks use much smaller blocks of data with higher spacial locality. Coupled with the use of TBB's cache allocators, this results in almost no misses for the L3 cache. Spawning and *get_task()* follow the same trend as those of Bodytrack because of the same reason: the *parallel_for* as well as the *parallel_reduce* templates are used together with the *grain size* parameter. Again, failed steals have the largest energy footprint among what we measured for this benchmark (see Fig. E.3).

Swaptions spawns over 600000 tasks, the largest number among all of our test applications. Like Bodytrack and Streamcluster, the *parallel_for* templates are prevented from dividing the workload too thin. In Fig. E.2 we can see that overheads grow with the core count which shows a higher parallel/sequential ratio than for the first 3 benchmarks. In Fig. E.3 we can see how failed stealing footprint grows significantly as the number of conflicts grows with the core count.

E.5.2 Improving task stealing

TBB uses a random victim selection policy. While fast and easy to implement, this approach is not fair: the same victim can be selected several times even if it is not the best

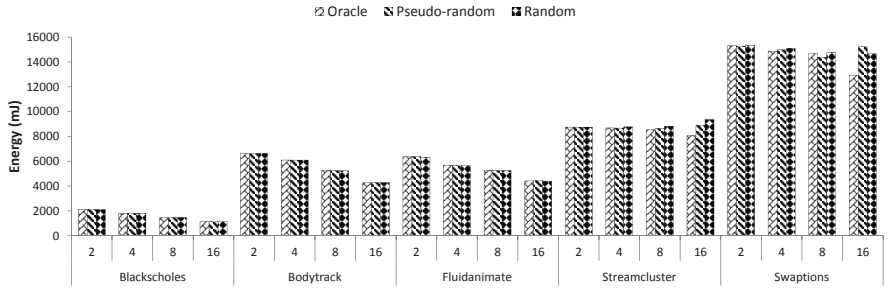


Figure E.5: Victim selection policies - comparison of total energy footprint

candidate [7]. Because we account part of the failed stealing attempts in our experiments to this exact scenario, we looked into changing the selection policy in order to improve performance and energy efficiency.

Fig. E.4 shows an overall decrease in the number of executed instructions for our oracle scheme. The results for both our victim selection methods in Fig. E.4 are normalized against TBB's random results. Benchmarks that have a small number of total tasks like Blackscholes, Bodytrack and Fluidanimate see only a marginal improvement in both Fig. E.4 and Fig. E.5. There are numerous phases during the execution of these benchmarks when all queues are empty. However, these phases are don't last enough to retire the workers meaning that they just waste energy trying to steal. Both Streamcluster and Swaptions show better results with the oracle selection in both Fig. E.4 and Fig. E.5.

Our results with the pseudo-random victim selection are also mixed. Overall we recorded an increase with all "bad" metrics like the number of failed stealing attempts, the number of conflicts and backoffs. However, the sequential scanning for victims is "cheaper" in terms of executed instructions than the default method which can be seen in Fig. E.4. Again, in terms of energy footprint Blackscholes, Bodytrack and Fluidanimate performed only marginally better or even worse than the default random selection. Streamcluster is the only benchmarked that showed improvements in all our test (see Fig. E.5). It is also worth mentioning that for 16-cores we recorded in average 2.14 times more failed stealing attempts than the default method. For Swaption, Fig. E.5 shows improvements for 2- to 8-cores but not for the 16-cores execution, where we recorded in average 4.27 times more failed stealing attempts than the default method. This shows that our pseudo-random implementation can be a bit too aggressive.

E.6 Related Work

The energy efficiency of parallel systems and the overheads parallelization brings have been the subject of many studies. Reducing the power requirements of multi-core CPUs, improving the energy efficiency of big parallel systems or reducing the overheads of parallel implementations have been explored by many researchers and plenty of solutions have been found. However, to the best of our knowledge, none of them tries to quantify the energy consumption of parallel overheads.

Li and Martinez studied the power-performance implications of running parallel applications on CMPs [14]. Using both an analytical model and detailed simulations, the authors show that parallel computing can bring significant power savings through judiciously selections of the granularity and voltage/frequency levels.

Contreras and Martonosi study and characterize some of the overheads of Intel's TBB [7]. They concluded that task management operation can have a detrimental effect on the performance of parallel execution. The authors also note that random stealing fails to scale with increasing core counts and that alternative policies can improve performance.

Bhattacharjee and Martonosi propose a thread criticality predictor which they build using memory hierarchy statistics [1]. The authors implement this predictor in two different applications. First, they implement it into TBB's task scheduler and show that task stealing can be improved over the original random approach. Second, they use the predictor to guide DVFS and to reduce dynamic energy in barrier-based applications. The authors conclude that the thread criticality predictor offers good accuracy at very low hardware overhead.

Podobas et al. do a performance comparative study of several TBP libraries, including TBB [20]. They use both micro-benchmarks and a subset of the BOTS suite to characterize application performance and the costs for task creation and stealing. The study concludes that Wool has the lowest overhead for task spawning and task stealing. However, our previous study showed Wool to be far more aggressive when stealing than TBB which means that as we scale up the core number, Wool will perform worse [12].

The *direct task stack* is a TBP algorithm for extremely fine grained parallel applications [9]. Its implementation in the Wool library shows very low overheads for task creation and task stealing. The experimental result show that Wool significantly outperforms other implementations like Cilk++, TBB or OpenMP for extremely fine grained parallel applications (tens of cycles/task).

Vandierendonck et al. advocate the use of TBP models with nested task spawning for writing general-purpose programs [22]. The authors developed a Cilk-like language to express parallel pipelines and extended a Cilk-like scheduler to recognize and enforce argument dependency types on task spawns. This programming model enhances the ease of programming parallel pipelines.

Chen et al. do a study to evaluate TBB's scalability against Pthreads implementations

and to measure some of TBB's overheads [5]. Their results show possible bottlenecks that limit the scalability of TBB. They also show that TBB runtime overheads increase with core counts and in the current implementation will become the main performance bottleneck when scaling to tens of cores.

Ami Marowka introduces TBBench, a micro-benchmark suite designed for Intel's TBB [16]. TBBench is designed to measure the overheads associated with *parallel_for* and *parallel_reduce* constructs and mutual exclusion mechanisms like *Mutex*, *Spin_mutex* and *Queuing_mutex*. The experimental results show that TBB's mutual exclusion mechanisms and scheduler exhibit less overheads than the equivalent OpenMP constructs.

E.7 Conclusion

Intel's TBB is a runtime library designed to encourage programmers to create portable, parallel applications using task parallelism. TBB was developed to dynamically scale on the existing resources, employing task stealing to deal with workload imbalance. Recently, the ICT sector is facing concerns about energy consumption and TBB has the potential of addressing these issues.

In the current study, we quantified the management overheads involved in parallelizing an application using TBB. We experimented with three victim selection policies. Using the "all knowing" oracle selection method, we saw a reduction of up to 60% in the number of executed instructions which translates into a 13% reduction in energy consumption compared to random victim selection. The pseudo-random also showed overall better results than the random scheme, with up to a 5% reduction in the energy footprint. We also looked at individual TBB operations like task spawning, task stealing and task dequeuing. Among these, we observed that the task stealing mechanism scales worst with core count and creates the highest energy footprint.

The results in this work suggest that there is a potential for improving the energy efficiency of victim selection policies. However, it is still unclear how to reach this potential with a practical implementation. We plan to investigate this and other issues in future work.

Bibliography

[1] A. Bhattacharjee and M. Martonosi. Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors. In *Proc. of the 36th annual Int'l Symp. on Computer Architecture*, 2009.

[2] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[3] S. Borkar and A. A. Chien. The Future of Microprocessors. *Commun. ACM*, 54(5), May 2011.

[4] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, Nov 2011.

[5] X. Chen, W. Chen, J. Li, Z. Zheng, L. Shen, and Z. Wang. Characterizing Fine-Grain Parallelism on Modern Multicore Platform. In *IEEE 17th Int'l Conf. on Parallel and Distributed Systems*, 2011.

[6] Cilk++. Cilk++: A quick, easy and reliable way to improve threaded performance. Available: `http://software.intel.com/en-us/articles/intel-cilk-plus/`. (retrieved November 2016).

[7] G. Contreras and M. Martonosi. Characterizing and Improving the Performance of Intel Threading Building Blocks. In *IEEE International Symposium on Workload Characterization*, pages 57–66, Sept 2008. doi: 10.1109/IISWC.2008.4636091.

[8] K.-F. Faxén. Wool - A Work Stealing Library. *SIGARCH Comput. Archit. News*, 36(5):93–100, June 2009. ISSN 0163-5964. doi: 10.1145/1556444.1556457. URL `http://doi.acm.org/10.1145/1556444.1556457`.

[9] K.-F. Faxén. Efficient Work Stealing for Fine Grained Parallelism. In *39th Int'l Conf. on Parallel Processing*, 2010.

[10] S. Fuller and L. Millett. Computing Performance: Game Over or Next Level? *Computer*, 44(1):31–38, Jan 2011. ISSN 0018-9162. doi: 10.1109/MC.2011.15.

[11] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval Simulation: Raising the Level of Abstraction in Architectural Simulation. In *Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 307–318, Feb. 2010.

[12] A. C. Iordan, M. Jahre, and L. Natvig. On the Energy Footprint of Task Based Parallel Applications. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 164–171, July 2013. doi: 10.1109/HPCSim.2013.6641409.

[13] D. Leijen, W. Schulte, and S. Burckhardt. The Design of a Task Parallel Library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640106. URL `http://doi.acm.org/10.1145/1640089.1640106`.

[14] J. Li and J. Martínez. Power-Performance Considerations of Parallel Computing on Chip Multiprocessors. *ACM Trans. Archit. Code Optim.*, 2, 2005.

[15] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multi-Core and Many-Core Architectures. In *International Symposium on Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM*, pages 469–480, Dec 2009.

[16] A. Marowka. TBBench: A Micro-Benchmark Suite for Intel Threading Building Blocks. *JIPS*, 8(2), 2012.

[17] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, Jan 2010. doi: 10.1109/HPCA.2010.5416635.

[18] H. Pan, B. Hindman, and K. Asanović. Composing Parallel Software Efficiently with Lithe. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2010.

[19] D. Patterson. The Trouble With Multicore. *IEEE Spectrum*, 47(7), 2010. ISSN 0018-9235.

[20] A. Podobas, M. Brorsson, and K.-F. Faxén. A Comparison of Some Recent Task-based Parallel Programming Models. In *Third Workshop on Programmability Issues for Multi-Core Computers*, 2009.

[21] TBB. Intel Corporation. *Intel Threading Building Blocks Reference Manual*. Available: `http://threadingbuildingblocks.org/`. (retrieved November 2016).

[22] H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos. Parallel Programming of General-Purpose Programs Using Task-Based Programming Models. In *Proc. of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, 2011.

Appendix F

Paper C.3

Tuning the Victim Selection Policy of Intel TBB

Alexandru C. Iordan, Magnus Jahre and Lasse Natvig

Journal of System Architecture (JSA)
2015

Abstract

The wide adoption of Chip Multiprocessors (CMPs) in almost all ICT segments has triggered a change in the way software needs to be developed. Parallel programming maximizes the performance and energy efficiency of CMPs, but also comes with a new set of challenges. Parallelization overheads can account for sub-linear speedups and can increase the energy consumption of applications. In past experiments we looked at specific operations such as spawning new tasks, dequeuing the task queue and task stealing for Intel TBB. Our results showed that failed steals account for the largest overhead. In this work, we focus on TBB's victim selection policy. We implement a new occupancy-aware policy and we improve the implementation of the pseudo-random policy we proposed in a previous paper. We compare the results of our new policies against an "oracle scheme" as well as against TBB's random victim selection approach. Our results show improvements in execution times and energy-efficiency of up to 11.23% and 14.72% respectively when compared to TBB's default policy.

F.1 Introduction

With Chip Multiprocessors present in almost any computing device today, software developers need to leverage the potential of this hardware and move towards parallel implementations. Parallel programming is a challenge mainly because there is no widely adopted programming model that facilitates easy parallelization. Parallel software development requires tools and methodologies to reduce time-to-market and maintenance effort. Over the last years, industry and academia have developed several parallel libraries that aim at improving application portability and programming efficiency [6, 13, 14, 21].

The introduction of CMPs almost a decade ago has enabled the mitigation of development constraints like the *power wall* and the *ILP wall* [8]. The performance potential of CMPs lies in exploiting thread level parallelism which means that parallel software is required to fully take advantage of this architecture. Intel's Thread Building Blocks (TBB) [21] is a runtime library designed to encourage software developers to create portable, parallel applications with task parallelism. TBB was developed to dynamically scale on the existing resources and employs task stealing to deal with workload imbalance. It was designed to allow developers to focus on parallelizing their code by providing a runtime system that handles parallelism management.

The cost of TBB's dynamic parallelism management is increased parallelization overhead. Developers may have to harness fine-grained parallelism from their applications in order to fully utilize a CMP's resources and this can incur high parallelization overheads. Understanding and limiting these overheads is a necessary step towards scalable and more efficient runtime parallel libraries. To this end, we investigate the extra instructions added by parallelization management and the energy consumption of these instructions which we refer to as the *energy footprint*. More precisely, the energy footprint is the energy spent for executing the given application or section of code in the context of the test system.

Our paper makes the following important contributions:

- We continue our study of the parallelism management costs of TBB [10, 11] and their impact on a CMP's energy efficiency. To allow for extensive and noninvasive measurements under increasing core counts, we use a performance simulator and a power estimation tool in our study.
- Extending our study into victim selection policies [11], we show that we can reduce thread contention and improve both execution times and the energy-efficiency of a parallel application when making an informed selection rather than a random one.
- We do a comparative study of several selection policies to show that with increasing core counts, the random victim selection policy employed by TBB is a serious performance bottleneck.

Our experiments show that parallelization overheads can cause sub-linear speedups leading to an increased energy consumption for parallel applications. In this paper, we look into mitigating the impact of these overheads and thereby reducing thread contention for

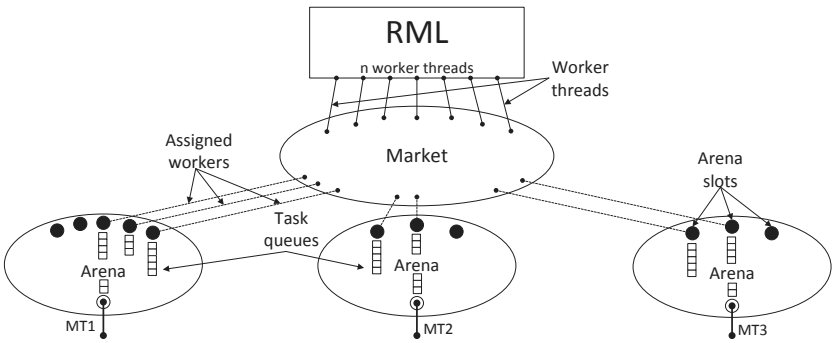


Figure F.1: Components of TBB's task scheduler

hardware resources. By changing TBB's random victim selection policy to an occupancy-aware or even to a pseudo-random policy we can achieve better performance or improved energy efficiency.

The paper is organized as follows: Section F.2 gives a general description of Intel TBB and its mechanisms for parallelizations. Section F.3 presents more details about the victim selection policy used in TBB as well as the policies we propose. The simulation tools and the benchmarks used in our experiments are described in Section F.4. In Section F.5, we present our study of the victim selection policies. Section F.6 presents the related work and Section F.7 concludes the paper.

F.2 Intel TBB

The concept of parallel programming is almost as old as the computer itself, yet it is a challenge for most developers. In today's *multi-core era* the overall efficiency of the system suffers if parallel applications are not developed to dynamically scale and take advantage of all the resources that are available to them. Over the years, many parallel languages have been developed and a multitude of research was done in an effort to improve performance and maximize hardware utilization [20]. With the majority of those approaches, one factor was often overlooked: the *composability* of the resulting solution. Composability of an application refers to its ability to run efficiently side by side with other applications and to be able to cope with the fact that it does not have exclusive access to the hardware resources [19]. We see this characteristic as a requirement for efficient exploitation of CMPs. For this reason, we focus on Intel's TBB version 4.1.1, which was designed to provide a high degree of composability.

Figure F.1 gives an overview of the structures TBB maintains in order to create and balance its parallel executing threads. The library allows parallelism to be annotated both

explicitly and implicitly. Explicit task creation is achieved through the use of methods like *spawn()* which gives the programmer complete control over the work performed by each task. Implicit task creation makes use of some templates like *parallel_for* or *parallel_reduce* which make code writing faster but gives control of the task creation over to the TBB library. Tasks are created and then added to the calling thread's task queue inside the *arena* (see Figure F.1). From the arena the task is available for execution by its owner thread or by other workers through stealing. A task can instantiate and spawn other tasks resulting in a hierarchical task tree.

A TBB *master thread* (*MT*) is an application thread that instantiates the *tbb::task_scheduler_init* object. All threads created by TBB to help complete the work of the MT are called *worker threads*. The *Resource Management Layer* (*RML*) is the component that hosts the pool of worker threads and gets instantiated first (see Figure F.1). No worker threads are created at this point, this being postponed until the first task is spawned.

Continuing top-down in Figure F.1, the *Market* is instantiated. This component was added in version 3.0 of TBB to ensure the composability of the framework. It separates the workload (the tasks) of one MT from other MTs that may be executing on the same machine. The role of the market is to assign workers to the arenas of each MT. The limit of the total number of workers available is set to 1 less than the maximum of the argument of the *tbb::task_scheduler_init* constructor and the total number of logical CPUs on the executing system.

The last structure to be created is the *Arena* associated with calling MT. An arena encapsulates all the tasks and the execution resources (worker threads) available to a MT. Each arena is assigned a number of slots representing the number of workers that arena requires to complete its parallel tasks. This is defined as 1 less than the minimum of the argument of the *tbb::task_scheduler_init* constructor and the total number of workers available (limit set by the market). Because several MTs can coexist, the total number of workers requested by all arenas can be greater than the number of workers available in the RML's pool. In this situation, the market will allot workers proportionally to each MT's request.

All these components and limits are created once, during the first instance of the *tbb::task_scheduler_init* object in the current execution. If an MT is not the first one to call the task scheduler, it will create a new arena that will comply with the limitation imposed by the market. Upon creation or destruction of an arena, the worker threads can migrate between the active arenas.

After they are created, each worker thread runs a scheduling procedure called *wait_for_all()* consisting of 3 nested loops. The inner loop executes the current task by calling its *execute()* method. TBB is a continuation-passing style library which means that the completion of this task returns a pointer to the next task that needs to be executed. If a new task is not referenced, the inner loop exits. In the middle loop the *get_task()* method tries to dequeue the local task queue in a LIFO order. If successful, the inner loop is called again. If unsuccessful because the queue is empty, the middle loop exits and the outer

loop invokes the stealing mechanism by calling the *receive_or_steal_task()* method.

F.3 The stealing mechanism

F.3.1 The TBB implementation

The *receive_or_steal_task()* method is part of the outer loop in the scheduling procedure and it looks for all work available at this level. This includes: tasks mailed via the task-to-thread affinity mechanism, reload offloaded non-priority tasks or reload tasks abandoned by other workers. If none of these calls return a task to execute, a steal is attempted from a randomly selected victim thread in the current arena. If the attempt is successful, the method returns and the scheduler re-enters the inner loop of the scheduling procedure. If unsuccessful, a failure counter is incremented and the execution pauses before looping back to the beginning of *receive_or_steal_task()* method. Also, if the failure counter surpasses a given threshold (default value is 100) and the arena is still empty, the current worker thread is freed and returns to the RML.

When attempting a steal, the thief must first get a lock on the victim's queue using the *lock_task_pool()* method. If that fails, the thief goes through a 5 step exponential backoff. After 5 fails, the current thread yields its resources and waits for its next time slot to try to lock the same victim again. This locking mechanism assures the high composability of TBB we discussed in Section F.2. However, the most common situation is when only one thread is running on each hardware core, making the yielding function return immediately. This means that the thief thread will continue trying to lock its victim. In our experiments, we match the simulated number of threads to the simulated number of cores which makes us face this locking issue.

The most common situation for stealing failure is due to selecting a victim with an empty task queue. Applications with an unbalanced workload distribution face this problem often. The default random selection policy in TBB cannot prevent against this type of failures.

Race contention is also a common situation for failure. When two or more threads are trying to get exclusive access to the same task queue by calling the *lock_task_pool()*, only one can succeed. A thief can return from the *lock_task_pool()* only if it either succeeds or the victim's task queue has been depleted. This means that the thread who did not acquire the lock will wait around until that lock is freed or until the victim queue has been emptied.

A special situation is when a thief thread is competing for access with the owner thread of that task queue. If there is more than one task in the queue, there is no race contention because the thief will steal at one end while the owner will dequeue the other. However, if there is only one task in the queue, the owner thread will have priority and the thief will backoff.

F.3.2 The oracle selection scheme

In an attempt to see how much performance can be improved by tuning the victim selection, we introduced an "all knowing" scheme we call the oracle selection [11]. This method leverages on the fact that we use a simulator and not a real machine. Thus, we can provide TBB with information that would be otherwise very "expensive" to obtain. Outside the simulated memory space, we created a data structure that stores the occupancy of each task queue in the arena as well as their level of congestion (the number of workers trying to steal from each queue). This structure is updated by the application through specialized instructions called markers that only our simulator recognizes and executes. Since we do all this computation outside the simulated environment, our TBB application sees the victim selection as an extremely fast, zero-overhead procedure. The scheme selects as victim the queue with some available tasks for stealing and with the lowest congestion level. Even though this oracle scheme provides very fast and accurate results, it is not perfect. For our simulator there are still a few situations when updates to our structure do not propagate fast enough and the selected victim ends up creating conflicts.

F.3.3 The pseudo-random selection scheme

Our second selection method is a pseudo-random scheme inspired by the Wool library [6]. This policy was also introduced in [11], but for this paper we improved its implementation and tuned its performance. For the first stealing attempt, we randomly select a task queue. If stealing from this victim fails, we then start a loop and sequentially scan the other active task queues, excluding the one of the current thread. In this way we will first try to steal from all possible queues before looping back in the *receive_or_steal_task()* and selecting a new random victim. There are two major benefits to this approach. First, all the stealing attempts during the sequential scan are very cheap in terms of number of instructions, reducing the overheads. Second, we can conclude much earlier than the TBB implementation that an arena is out of work and we can put a worker thread to sleep sooner. To tune our implementation even further, we removed the call to the yielding function from the *lock_task_pool()*. This forces the method to return after the 5 steps exponential backoff and eliminates the conflicts caused by the immediate return of the yielding function. However, this makes the stealing mechanism a bit more aggressive since it allows it to select new victims faster.

F.3.4 The occupancy-aware selection scheme

This method is inspired by the oracle scheme and tries to find the task queue with the most work available to steal from. In contrast to the oracle scheme, we now select our victim based solely on the level of occupancy of the task queues. Also, in contrast to our "all knowing" policy, this scheme is implemented fully in the TBB library and can

Table F.1 Main characteristics of modeled processor

	Parameter	Value	
Core	Clock frequency	2.66 GHz	
	Instruction set	x86-64	
	Dispatch width	4	
	Window size	128	
	Core count	1,2,4,8,16,32 cores	
Cache		Size	Assoc.
	L1 iCache/dCache	#cores x 32KB	4/8
	L2 Cache	#cores x 256KB	8
	L3 Cache	2/4/8/16/32/64 MB	16
Main memory	Size	2/4/8/16/32/64 GB	

be used outside of our simulated environment. We use a 2-dimensional array to store the occupancy level of the queues, with each thread logging separately information about tasks that it spawned, tasks that it stole or tasks that it executed. In this way we eliminate the possibility of races on writing and the need for a locking mechanism. To increase selection speed, we also do the scanning of the array with no locks. All these ensure that this approach is fast enough to work with TBB. However it also means that a snapshot of the occupancy array will not always be accurate. Since the congestion level of the task queues are not monitored (like the oracle policy does), a queue can be selected as victim by several thieves at the same time. To make sure the thieves will first deplete the tasks of this victim before attempting a new selection, we used the default TBB approach for the *lock_task_pool()* function. A thief will not return from this function unless it either acquired the lock or the task queue is empty. With this selection scheme, just like with the pseudo-random one, we can find out faster than the default TBB approach that an arena is out of work.

F.4 Methodology

F.4.1 Simulation tools

We performed our experiments using a parallel, x86 computer architecture simulator called Sniper [3]. Sniper uses the interval core model [9] and Graphite simulation infrastructure [18] to provide fast and accurate simulations. Our model is a Nehalem-based Xeon 5500-series multi-core CPU (code name Gainestown) with a clock frequency of 2.66 GHz and 3 levels of cache. The simulations do not include an operating system and no mechanism for frequency and/or voltage control is used. Table F.1 lists the main characteristics of the modeled processor.

The performance results from Sniper are fed into a power estimation tool called McPAT [16]. An important characteristic of McPAT is its ability to model dynamic, static and short-circuit power. Dynamic power refers to the power required by a circuit to switch from one logical state to the other. For each system component, dynamic power is defined as: $power_{dynamic} \sim AF \cdot C \cdot V_{dd}^2 \cdot f$, where *AF* is the activity factor, *C* is the total load capacitance, V_{dd} is the supply voltage and f is the clock frequency [12]. Switching circuits also dissipate short-circuit power which McPAT modeles analytically. Static power is caused by current leakage during periods of non-activity. McPAT estimates leakage current using models of real-world CMOS circuits.

A recent study shows that McPAT's area and power models can have significant errors [24]. The authors assess McPAT's estimations against measurements of an IBM POWER7 CMP. They note that read/write port overestimates caused by high issue width and modeling of simultaneous multithreading (SMT) are two of the major sources of error they observed. For this reason, our measurements are only marginally affected by these errors since our modeled CPU has a relative low issue width (4 compared to 8 for the POWER7) and no SMT enabled.

To account for both active and idle core time, we use the dynamic power and static power (totaling subthreshold and gate leakage) outputted by McPAT for each core. In estimating the energy footprint, we multiply these by the active runtime and the idle time respectively of the cores to get a measure of the energy they use. Adding them all together gives us the CPU energy usage.

F.4.2 Benchmarks

For our experiments, we used the default TBB implementations of *Blackscholes*, *Bodytrack*, *Fluidanimate*, *Streamcluster* and *Swaptions* benchmarks with the simlarge input set from the PARSEC suite [2]. All of them were built using the 4.1.1 version of TBB. Collectively, these benchmarks express parallelism both explicitly as well as implicitly and employ some special TBB constructs like cache affinity partitioners and cache allocators. They provide a wide test base for our study.

Blackscholes uses the Black-Scholes partial differential equation to analytically calculate the prices for a portfolio of European options. The differential equation is implemented numerically and *parallel_for* templates are employed to divide the work among worker threads. In order to improve cache affinity, a TBB affinity partitioner is used.

Bodytrack is a computer vision application which tracks a human body with multiple cameras. It uses pipeline parallelism and *parallel_for* templates to express parallelism.

Fluidanimate simulates an incompressible fluid for interactive animation purposes. It uses an extension of the Smoothed Particle Hydrodynamics method to describe the fluid. Parallelism is annotated explicitly through *spawn(task_list)*.

Streamcluster is a mining application that tries to solve the online clustering problem.

Parallelism is annotated explicitly through *spawn(task_list)* as well as using *parallel_for* and *parallel_reduce* templates. TBB's cache allocators are also used to optimize access to shared data.

Swaptions uses the Heath-Jarrow-Morton framework to price a portfolio of swaptions. Price computation is achieved through the Monte Carlo simulation. Swaptions uses *parallel_for* templates and cache allocators to express parallelism and optimize access to shared data.

Our experiments are meant to study the impact of the victim selection policy on the overall performance of the parallel execution. To that end, we want to minimize all possible interference on our test policies and quantify their impact as accurately as possible. To eliminate context switching on the simulated cores, we always match their number with the number of parallel threads. Also, we simulated only one benchmark at a time. It will be very difficult (if not impossible) to account for the effects of thread interleaving when two or more applications are executed at the same time.

To account for the non-deterministic simulation of Sniper, we performed 10 simulations of each benchmark for every core count. We averaged the performance results (μ) and used these to estimate the power requirements. We also computed the standard deviation (σ) of the execution time for each of the 10 simulation set. In none of our experiments we found any outliers, where an outlier is a value beyond $3\sigma \pm \mu$. For Blackscholes, Bodytrack, Fluidanimate and Streamcluster, our results show a σ/μ in the 0.012% - 1.83% range. Swaptions, due to its use of the Monte Carlo simulation has a higher variability between simulation, with σ/μ in the 1.18% - 14.73% range.

F.5 Results

As described by Amdahl's law, the maximum expected speedup of parallelization is limited by the sequential fraction of the program. When managing overheads are taken into consideration, this theoretical maximum becomes even harder to achieve. As we showed in our previous study, these overheads become larger as we scale the core count [11]. Even though with parallel executions the work gets done faster, the energy required to complete it is often equal or greater than the sequential execution.

F.5.1 Parallelization overheads

As mentioned in Section F.2, each worker thread runs a scheduling procedure containing an infinite nested loop. This loop tries to execute tasks from its own queue or to obtain some work through the *receive_or_steal_task()* method. By default, the *receive_or_steal_task()* method loops a maximum of 100 times in an attempt to obtain a task before reporting that the arena is empty and returning the thread to RML. This means that each time

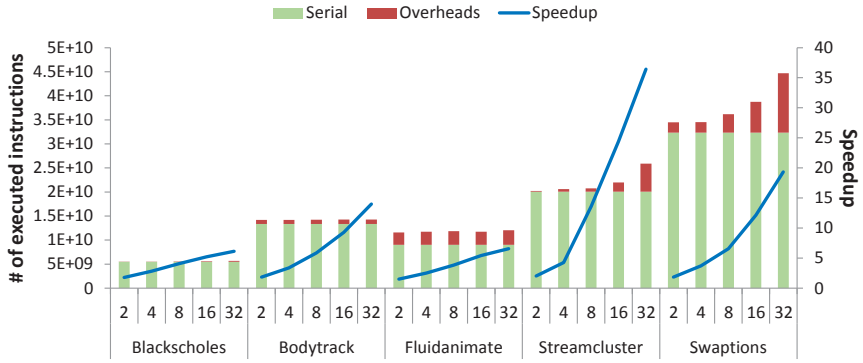


Figure F.2: Overheads and speedups for the default random selection policy

a steal fails, the *receive_or_steal_task()* will loop to the beginning adding overheads and delay to the execution.

A very simple way to see what trend parallelization overheads form as you scale up the number of cores is to look at the execution statistics reported by TBB. There you can see how many times each parallel thread successfully stole a task, how many times it failed, how many times out of those fails was due to conflicts with other threads and many other. Looking at these statistics for the default TBB implementation, it becomes apparent that random victim selection policy is a serious bottleneck for high core counts. For applications with high numbers of parallel tasks like Swaptions, failed tasks range from an average of 40000 for 2-cores executions to almost 18 millions for 32-cores ones. That translates into 38.18% increase in instruction count when compared to the serial execution (see Figure F.2). A detailed analysis of the results presented in Figure F.2, including a breakdown of the overheads and a discussion on speedups, can be found in [11].

With the occupancy-aware selection scheme we wanted to reduce the overheads by removing all (or as many as possible) failed tasks caused by conflicts. Although we managed to do that, the overall overheads are generally higher than those of the random selection experiments. This is due to the fact that we scan the occupancy array for each steal attempt and this adds up fast. For all our low core counts (2 or 4) results there is not enough contention among threads in order to balance-out the added number of instructions of the scanning operation. In addition, some benchmarks like Blackscholes, Bodytrack and Fluidanimate have low numbers of total tasks to execute which again makes it hard to make up for the overhead of the scanning operation.

In the case of the pseudo-random selection policy, things are almost the opposite of occupancy-aware scheme: we generally have more failed steal attempts, but overall the

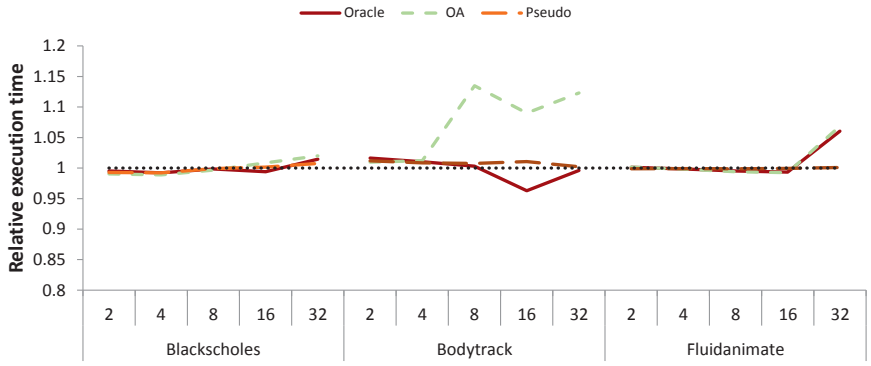


Figure F.3: Total execution times relative to the random selection policy

overheads are lower. This is explained by the fact that the pseudo-random policy is far more aggressive in trying to find new work, but due to our sequential scanning implementation each attempt is cheaper. Also, the *receive_or_steal_task()* method returns much faster reducing the overheads even further.

F.5.2 Victim selection policies - comparative study

The main issue faced by the random selection policy is its inability to scale. For high core counts or when we are dealing with very fine parallelism which forces the worker threads to steal often, random selection causes overheads to grow exponentially. We developed the occupancy-aware and the pseudo-random schemes to address this limitation, by adding some information gathering in the selection process. By doing this we increased the work the threads need to do, so the added performance has to pay for this as well. Because of its very simple nature, the random victim selection policy remains hard to outperform in situations when race contention among threads are rare (see Figure F.3). Our occupancy-aware policy proves to be great in theory but difficult in practice. Our results show that it manages to significantly reduce the conflicts among threads. However, our implementation relies on scanning the occupancy array for each steal attempt which proves to be very costly. In addition, we implemented some guards against conflicts with the main thread which proved to have unexpected effects in some situations (see the 2-core results for Streamline in Figure F.4). What becomes apparent when looking at the results in Figure F.3 and F.5 is that we can't always afford the added complexity. However, when there is enough congestion for this policy to make a difference, it can reduce execution time with up to 11.23% and the energy footprint with up to 7.83% (see Figure F.4 and F.6).

The pseudo-random selection is much lighter in terms of extra-work compared to the

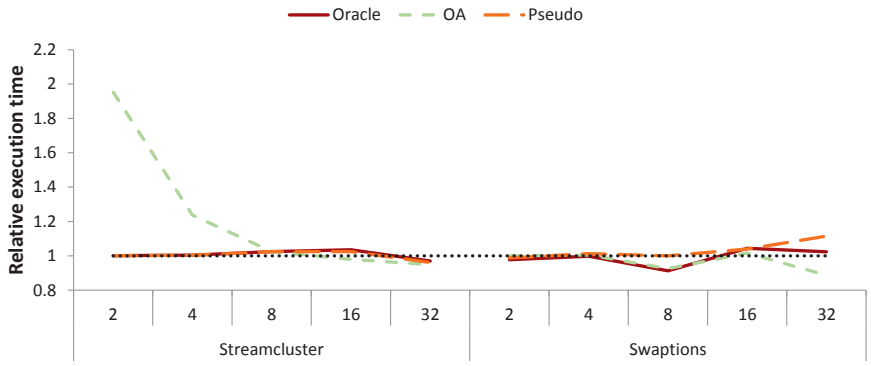


Figure F.4: Total execution times relative to the random selection policy

occupancy-aware policy, but is also more aggressive. Our results with this scheme show that it can only be marginally faster than the default random selection, but it constantly does better in terms of energy-footprint (see Figure F.3, F.4, F.5 and F.6). This happens because with this policy it is very easy to identify the situations when there is no work to be done by the worker threads. By putting them to sleep sooner, we save energy. In this way it manages to reduce the energy footprint with up to 14.72%.

F.6 Related Work

The energy efficiency of parallel systems and the overheads parallelization brings have been the subject of many studies. Reducing the power requirements of multi-core CPUs, improving the energy efficiency of big parallel systems or reducing the overheads of parallel implementations have been explored by many researchers and plenty of solutions have been found.

Li and Martinez studied the power-performance implications of running parallel applications on CMPs [15]. Using both an analytical model and detailed simulations, the authors show that parallel computing can bring significant power savings through judiciously selections of the granularity and voltage/frequency levels.

Contreras and Martonosi study and characterize some of the overheads of Intel's TBB [5]. They concluded that task management operation can have a detrimental effect on the performance of parallel execution. The authors also note that random stealing fails to scale with increasing core counts and that alternative policies can improve performance.

Bhattacharjee and Martonosi propose a hardware thread criticality predictor which they build using already-accessible on-chip information like memory statistics [1]. The au-

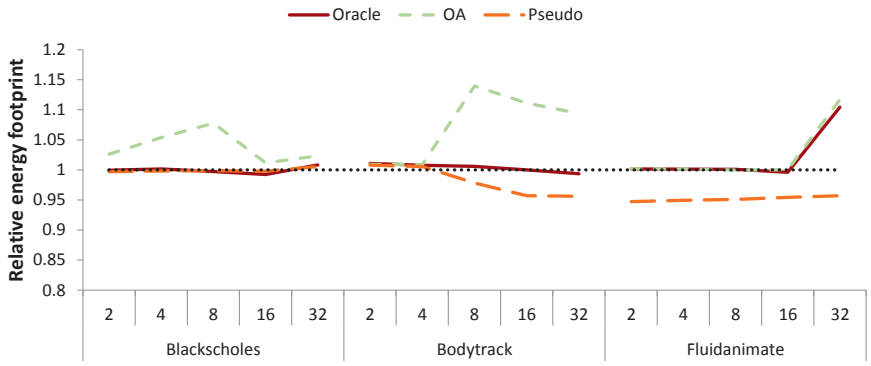


Figure F.5: Energy footprint relative to the random selection policy

thors test this predictor in two different scenarios. First, they use it to assist TBB's task scheduler and show that task stealing can be improved over the original random approach. Second, they use the predictor to guide DVFS and to reduce dynamic energy in barrier-based applications. The authors conclude that the thread criticality predictor offers good accuracy at very low hardware overhead.

Podobas et al. do a performance comparative study of several parallelization libraries, including TBB [22]. They use both micro-benchmarks and a subset of the BOTS suite to characterize application performance and the costs for task creation and stealing. The study concludes that Wool has the lowest overhead for task spawning and task stealing. However, our previous study showed Wool to be far more aggressive when stealing than TBB which means that as we scale up the core number, Wool will perform worse [10].

The *direct task stack* is a TBP algorithm for extremely fine grained parallel applications [7]. Its implementation in the Wool library shows very low overheads for task creation and task stealing. The experimental results show that Wool significantly outperforms other implementations like Cilk++, TBB or OpenMP for extremely fine grained parallel applications (tens of cycles/task).

Vandierendonck et al. advocate the use of TBP models with nested task spawning for writing general-purpose programs [23]. The authors developed a Cilk-like language to express parallel pipelines and extended a Cilk-like scheduler to recognize and enforce argument dependency types on task spawns. This programming model enhances the ease of programming parallel pipelines.

Chen et al. do a study to evaluate TBB's scalability against Pthreads implementations and to measure some of TBB's overheads [4]. Their results show possible bottlenecks that limit the scalability of TBB. They also show that TBB runtime overheads increase with core counts and in the current implementation will become the main performance

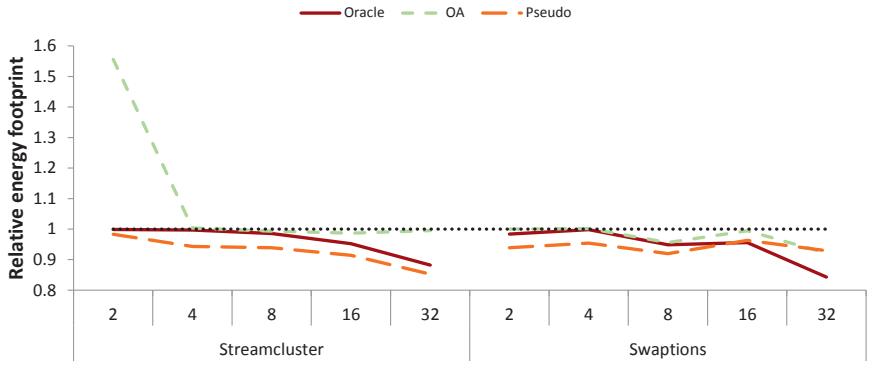


Figure F.6: Energy footprint relative to the random selection policy

bottleneck when scaling to tens of cores.

Ami Marowka introduces TBBench, a micro-benchmark suite designed for Intel's TBB [17]. TBBench is designed to measure the overheads associated with *parallel_for* and *parallel_reduce* constructs and mutual exclusion mechanisms like *Mutex*, *Spin_mutex* and *Queuing_mutex*. The experimental results show that TBB's mutual exclusion mechanisms and scheduler exhibit less overheads than the equivalent OpenMP constructs.

F.7 Conclusion

Intel's TBB is a runtime library designed to encourage programmers to create portable, parallel applications using task parallelism. TBB was developed to dynamically scale on the existing resources, employing task stealing to deal with workload imbalance. However, as CPU's core counts are ever-increasing, TBB proves to have a performance bottleneck in its use of a random victim selection policy.

Continuing our previous study [11], we propose two alternatives for the victim selection process. Based on the "all knowing" oracle scheme, we developed an occupancy-aware policy to reduce the number of failed steals. However, our implementation proved to be too complex and in many situation we recorded an overall increase in overheads. Nevertheless, for applications with very high thread contention, this scheme proved to be very beneficial, reducing the execution time and the energy footprint with up to 11.23% and 7.83% respectively. We think that our implementation can be improved and we will pursue this in future work.

The pseudo-random victim selection is the second policy we experimented with. The implementation in this paper is a refinement of the one in [11] and it showed better energy

footprints across the board when compared to the default TBB scheme. Even though it copes better in situations with many races between threads than the random one, the pseudo-random selection's performance is still affected in such scenarios.

With this work we showed that TBB can be improved for both performance and energy efficiency, even though not always at the same time. The results of our occupancy-aware scheme can be improved and we plan to do this in future work. Also, seeing how the pseudo-random approach performs well under low core counts, we are also considering a combined selection policy. The idea is to use each scheme for the core counts that they perform best. Based on our experiments so far, for core counts of 2 to 8 pseudo-random could be used and occupancy-aware for anything above. However, a more extensive testing needs to be done on a larger number of benchmarks before confirming this threshold.

Bibliography

[1] A. Bhattacharjee and M. Martonosi. Thread Criticality Predictors for Dynamic Performance, Power, and Resource Management in Chip Multiprocessors. In *Proc. of the 36th annual Int'l Symp. on Computer Architecture*, 2009.

[2] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[3] T. E. Carlson, W. Heirman, and L. Eeckhout. Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–12, Nov 2011.

[4] X. Chen, W. Chen, J. Li, Z. Zheng, L. Shen, and Z. Wang. Characterizing Fine-Grain Parallelism on Modern Multicore Platform. In *IEEE 17th Int'l Conf. on Parallel and Distributed Systems*, 2011.

[5] G. Contreras and M. Martonosi. Characterizing and Improving the Performance of Intel Threading Building Blocks. In *IEEE International Symposium on Workload Characterization*, pages 57–66, Sept 2008. doi: 10.1109/IISWC.2008.4636091.

[6] K.-F. Faxén. Wool - A Work Stealing Library. *SIGARCH Comput. Archit. News*, 36(5):93–100, June 2009. ISSN 0163-5964. doi: 10.1145/1556444.1556457. URL `http://doi.acm.org/10.1145/1556444.1556457`.

[7] K.-F. Faxén. Efficient Work Stealing for Fine Grained Parallelism. In *39th Int'l Conf. on Parallel Processing*, 2010.

[8] S. Fuller and L. Millett. Computing Performance: Game Over or Next Level? *Computer*, 44(1):31–38, Jan 2011. ISSN 0018-9162. doi: 10.1109/MC.2011.15.

[9] D. Genbrugge, S. Eyerman, and L. Eeckhout. Interval Simulation: Raising the Level of Abstraction in Architectural Simulation. In *Proceedings of the 16th IEEE Inter-*

national Symposium on High-Performance Computer Architecture (HPCA), pages 307–318, Feb. 2010.

[10] A. C. Iordan, M. Jahre, and L. Natvig. On the Energy Footprint of Task Based Parallel Applications. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 164–171, July 2013. doi: 10.1109/HPCSim.2013.6641409.

[11] A. C. Iordan, M. Jahre, and L. Natvig. Victim Selection Policies for Intel TBB: Overheads and Energy Footprint. In E. Maehle, K. Römer, W. Karl, and E. Tovar, editors, *Architecture of Computing Systems - ARCS 2014*, volume 8350 of *Lecture Notes in Computer Science*, pages 13–24. Springer International Publishing, 2014. ISBN 978-3-319-04890-1. doi: 10.1007/978-3-319-04891-8_2. URL http://dx.doi.org/10.1007/978-3-319-04891-8_2.

[12] S. Kaxiras and M. Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan and Claypool Publishers, 2008.

[13] D. Leijen, W. Schulte, and S. Burckhardt. The Design of a Task Parallel Library. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 227–242, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640106. URL http://doi.acm.org/10.1145/1640089.1640106.

[14] C. E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010. ISSN 0920-8542. doi: 10.1007/s11227-010-0405-3. URL http://dx.doi.org/10.1007/s11227-010-0405-3.

[15] J. Li and J. Martínez. Power-Performance Considerations of Parallel Computing on Chip Multiprocessors. *ACM Trans. Archit. Code Optim.*, 2, 2005.

[16] S. Li, J. Ahn, R. Strong, J. Brockman, D. Tullsen, and N. Jouppi. McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multi-Core and Many-Core Architectures. In *International Symposium on Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM*, pages 469–480, Dec 2009.

[17] A. Marowka. TBBench: A Micro-Benchmark Suite for Intel Threading Building Blocks. *JIPS*, 8(2), 2012.

[18] J. E. Miller, H. Kasture, G. Kurian, C. Gruenwald, N. Beckmann, C. Celio, J. Eastep, and A. Agarwal. Graphite: A Distributed Parallel Simulator for Multicores. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, Jan 2010. doi: 10.1109/HPCA.2010.5416635.

[19] H. Pan, B. Hindman, and K. Asanović. Composing Parallel Software Efficiently with Lithe. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2010.

[20] D. Patterson. The Trouble With Multicore. *IEEE Spectrum*, 47(7), 2010. ISSN 0018-9235.

[21] C. Pheatt. Intel Threading Building Blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, Apr. 2008. ISSN 1937-4771. URL `http://dl.acm.org/citation.cfm?id=1352079.1352134`.

[22] A. Podobas, M. Brorsson, and K.-F. Faxén. A Comparison of Some Recent Task-based Parallel Programming Models. In *Third Workshop on Programmability Issues for Multi-Core Computers*, 2009.

[23] H. Vandierendonck, P. Pratikakis, and D. S. Nikolopoulos. Parallel Programming of General-Purpose Programs Using Task-Based Programming Models. In *Proc. of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, 2011.

[24] S. L. Xi, H. Jacobson, P. Bose, G.-Y. Wei, and D. Brooks. Quantifying sources of error in McPAT and potential impacts on architectural studies. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 577–589, Feb 2015. doi: 10.1109/HPCA.2015.7056064.