An Adaptive Middleware for Improved Computational Performance

Bonnichsen, Lars Frydendal

Publication date:
2016

Document Version
Publisher's PDF, also known as Version of record

An Adaptive Middleware for Improved Computational Performance

Lars Frydendal Bonnichsen

Kongens Lyngby 2015

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary (English)

The performance improvements in computer systems over the past 60 years have been fueled by an exponential increase in energy efficiency. In recent years, the phenomenon known as *the end of Dennard's scaling* has slowed energy efficiency improvements — but improving computer energy efficiency is more important now than ever. Traditionally, most improvements in computer energy efficiency have come from improvements in lithography — the ability to produce smaller transistors — and computer architecture - the ability to apply those transistors efficiently. Since the end of scaling, we have seen diminishing returns from developments in lithography and modern computer architectures are so complicated requiring significant programming effort to exploit efficiently — software developers undertaking such a task will need all the help they can get, in order to keep the programming effort down.

In this thesis we champion using software to improve energy efficiency — in particular we develop guidelines for reasoning and evaluating software performance on modern computers, and a middleware that has been designed for modern computers, improving computational performance both in terms of energy and execution time. Our middleware consists of a new power manager, synchronization libraries using hardware transactional memory (for locks, barriers, and task synchronization), and two concurrent map data structures, which can be deployed in computer systems with little to no effort. At a fundamental level, we are improving computational performance by exploiting modern hardware features, such as dynamic voltage-frequency scaling and transactional memory. Adapting software is an iterative process, requiring that we continually revisit it to meet new requirements or realities; a time consuming process which we hope to simplify by analyzing the realities of modern computers, and providing guidelines explaining how to get the most performance out of them.

Summary (Danish)

De sidste 60 år er computere blevet drastisk hurtigere, takket være eksponentielle forbedringer i deres energi effektivitet. Energi forbedringerne har sænket farten de sidste år, pga. et fænomenet *the end of Dennard's scaling*, på trods af at det i dag er ekstremt økonomisk attraktivt at lave energi effektivitet. Traditionelt set har de største forbedringer i beregningsmæssig energi effektivitet kommet fra litografi, som har leveret mindre transistorer, og computer arkitektur, som har anvendt det stigende antal transistorer effektivt. Desværre giver mindre transistorer aftagende afkast, og forbedringerne i computer arkitektur har gjort computere så komplekse at det ofte kræver en ekstrem programmerings indsats at anvende dem effektivt. Hvis moderne software udviklere skal kunne udnytte moderne computer arkitekturer effektivt, så behøver de al den hjælp de kan få.

I denne afhandling bruger vi software til at forbedre energi effektivitet, ved (1) at udvikle retningslinjer til at optimere og evaluere software ydelse, baseret op analyser af moderne computere, og (2) at optimere middleware mod moderne computeres features, såsom dynamisk skift af spænding og frekvens og transaktionel hukommelse. Vores middleware består af en ny strømstyring, synkroniserings biblioteker som bruger hardware transaktionel hukommelse (til låse, barriere, og tasks), og to parallelle data strukturer (maps). Alle delene af middlewaren kan genanvendes med relativt lille arbejdsindsats. At tilpasse software er i sig selv en iterativ proces, som kræver at vi genovervejer vores fremgangsmåder, så vi kan optimere mod nye krav og nye realiteter. Det er en tidskrævende proces, som vi håber på at forenkle igennem retningslinjer, som forklarer hvordan vi bedst udnytter moderne computere.

Preface

This thesis was prepared at the Department of Applied Mathematics and Computer Science, in the Technical University of Denmark in partial fulfillment of the requirements for acquiring the Ph.D. degree in engineering.

The Ph.D. study was carried out under supervision of Professor Christian W. Probst and Professor Sven Karlsson in the period from October 2012 to September 2015.

The Ph.D. project was funded by the European collaborative project PaPP (Portable and Predictable Performance on Heterogeneous Embedded Manycores) funded jointly by the ARTEMIS Joint Undertaking and national governments under the Call 2011 Project Nr. 295440.

Kongens Lyngby, September 2015
Lars Frydendal Bonnichsen

Acknowledgements

First I would like to thank my supervisors, Christian and Sven. Their support and direction during the development of this thesis helped me better understand and express the problems the problems I have faced. I would especially like to thank them for fostering an environment that encourages diverse approaches, where no solution is off the table.

I am grateful to my defense committee, Alberto Lluch Lafuente, Uwe Assmann, and Rene Rydhof Hansen, for their interest, inquiries, and pleasant demeanor.

I would like to thank Alessandro, Artur, Andreas, Laust, Marieta, Nicklas, and my supervisors, for assisting in proofreading various drafts of this thesis. Their help was instrumental for providing some semblance of a readable manuscript. They cannot be blamed for any remaining faults or poorly constructed phrases — those are entirely my doing.

I am deeply appreciative of Vladimir Vlassov, and the group at the School of Information and Communication Technology at KTH, for hosting me on an external stay. I am especially thankful for the warm welcome that I received from Ahsan, Ananya, Artur, and Georgios during the stay, and the many productive, enlightening, and memorable discussions we have had.

I am forever indebted to the Winamp team, especially Justin Frankel, for creating the Advanced Visualization Studio — a creative and friendly development environment — which was my first real entry into programming.

I would like to thank all my friends at DTU and all my friends outside work,

especially Martin and Kavan, for all the memorable moments, and for providing a mostly stress free environment where I could regain my sanity, without worrying about the state of computer science.

Finally, I would like to thank my family, and especially my parents, for supporting me throughout my life, and in particular throughout the past three years. If you are reading this, please know that I love all of you, and cannot thank you enough for all you have done for me.

Contents

Chapter 1

Introduction

It seems to be popular opinion that writing efficient programs will be increasingly difficult and expensive in the coming years, as programmers have to come up with increasingly contrived ways of exploiting available hardware. With this thesis I will show that it is possible to improve computational performance — both in terms of *energy* and *execution time* efficiency — by adapting systems' middleware layer to current workloads and hardware trends.

Improving energy efficiency is one of the main challenges for modern computer systems. In 2012 computers were one of the main electricity consumers, representing an estimated 4.6 % of the world's electricity consumption [VHLL+14] — a ratio which seems to be ever increasing [Koo08, LVHV+12, Mil13]. Reducing computer energy consumption will not only reduce the economical and environmental costs associated with computing, b½ut also allow for faster computers, since most modern computers are designed to fit within a fixed power budgets. Technologies which improve computational speed can also be applied to improve computational energy efficiency, so from our perspective, both.

At a hardware level, computational speed is usually limited by feasible power consumption. Increasing power consumption reduces battery life and generates more heat. The heat generated in modern data centers results in expensive cooling requirements, which typically consume as much energy as the computers themselves. In short, improving energy efficiency makes computation cheaper,

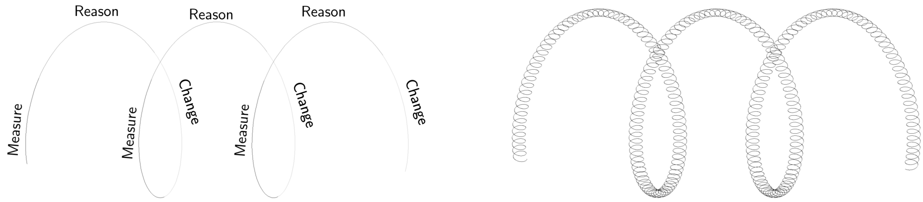
faster, less harmful to the environment, and easier to power by batteries.

Traditionally, computers have become exponentially more energy efficient, largely thanks to improvements in computer architecture and lithography [KBSW11]. The improvements in lithography directly resulted in smaller and more energy efficient transistors. The improvements in computer architecture have found ways to apply the increasing number of transistors efficiently. Unfortunately the improvements from both computer architecture and lithography have decreased over recent years, in a phenomenon known as *the end of Dennard's scaling*, which is characterized by slower performance improvements from the technology for developing and producing smaller transistors (lithography), and the technology for applying an increasing number of transistors (computer architecture).

Developments in lithography have grown increasingly complicated and expensive [M+11, HH08]. This trend is not new, but the developments have slowed down in recent years, as the size of transistors' gate oxide approximates the size of atoms, making them more susceptible to quantum mechanical effects. The lower limit viable transistor size has been adjusted several times with unexpected technological breakthroughs [BMSMMA12], but at this moment even Intel expects transistor size reductions to be slower in the foreseeable future [Alp15]. Meanwhile, the energy efficiency benefits from smaller transistors have lessened, because the power consumption of smaller transistors is more affected by leakage power [Boh07].

Computer architectures have had difficulties in efficiently using the increasing number of transistors available on computer chips. Since the 1980's most computer architecture improvements have come from exploiting parallel computations. Some computations can be performed in parallel implicitly, while other computations must explicitly be made parallel by programmers. Implicit parallelism is more attractive, because it does not require more programming effort, but it has also yielded diminishing returns.

The end of scaling means that it has become increasingly difficult and expensive to improve computational performance in terms of execution time and energy efficiency, increasing the value of alternative approaches. From a software standpoint, we can improve computation performance through (1) improved power management, and (2) faster algorithms, particularly algorithms which expose parallelism efficiently. All energy efficiency improvements in software require some effort to apply, but reusable software components, or middleware, can improve the energy efficiency of a wide range of applications relatively cheaply. My main hypothesis is:



(a) The adaptive procedure iterated three times.

(b) An adaptive procedure nested within an adaptive procedure.

Figure 1.1: The adaptive procedure.

> Middleware adapted to current needs can improve computational performance without being prohibitively expensive to deploy.

If the hypothesis holds, such middleware will become increasingly attractive due to diminishing energy efficiency improvements from lithography and computer architecture. We test the hypothesis by developing a series of such $\frac{111111111111111111111111111111111111111111111111}{222222222222222222222222222222222222222222222222}$, which require little to no deployment effort. The middleware are all developed in a common adaptive mindset, illustrated by Figure 1.1a: We measure, reason about, and change our programs to optimize them:

- Measuring their execution time, memory allocations, and general behavior.
- Reasoning about the programs bottlenecks, and how we can remove them.
- Changing the programs to remove the bottlenecks.

It is an iterative process, which can be repeated several times. The adaptive process can also be automated, such that the computer dynamically measures, detects bottlenecks, and changes its own behavior accordingly. Both automatic and manual adaptivity can coexist in the same system, as illustrated by Figure 1.1b. For instance, we can manually adapt an algorithm in a program which is running on a computer which automatically adapts to its workloads. Having such nested adaptive processes will affect how they change and reason about their bottlenecks, hopefully complementing each other. Manually adapting systems in the adaptive process mirrors the scientific model — stating a hypothesis (reasoning), testing the hypothesis (measuring), and accepting or rejecting the hypothesis (changing). Automatically adapting systems in the adaptive process mirrors the general structure of control systems — consisting of sensors, which measure the environment, actuators which change the environment, and control logic which decides which changes to make (reasoning).

1.1 Contributions

Through the adaptive approach — measuring, reasoning, and changing — we have developed a series of new middleware which can significantly improve performance:

1. A power management scheme `pappeadapt`, which reduces energy consumption by adapting the processors voltage and frequency settings to accommodate the current workload.
    - On four benchmark suites, `pappeadapt` reduced energy consumption by 41.5 % (geometric mean), without requiring any changes to the benchmarks.
2. New implementations of lock and barriers which minimize the amount of time threads spend waiting for each other (blocking), by applying modern computers support for Hardware Transactional Memory (HTM).
    - The lock and barrier implementations enable applications to benefit further from explicit parallelism, without requiring any changes except that the applications link against our OpenMP library (TurboBŁYSK).
3. BT-trees, a new efficient implementation of map data structures, providing high sequential and parallel performance on modern computers with HTM support.
    - operations on BT-trees are up to 5.3 times faster than traditional maps using HTM, and up to 3.9 times faster than state of the art concurrent ordered maps.
4. ELB-trees, a new implementation of relaxed map data structures, providing high parallel performance in workloads which support its relaxed semantics without requiring HTM support.

Our findings have contributed to deliverables in the PaPP project, as well as the following publications:

- Lars Bonnichsen, Sven Karlsson, Christian Probst: "ELB-trees an efficient and lock-free B-tree derivative" in the International Workshop on Multi-/Many-code Computing Systems (*MuCoCoS*) 2013, IEEE [BKP13]. This paper introduces and evaluates ELB-trees.

- Lars Bonnichsen, Christian Probst, Sven Karlsson: "Hardware Transactional Memory Optimization Guidelines, Applied to Ordered Maps" in the International Symposium on Parallel and Distributed Processing with Applications (*ISPA*) 2015, IEEE [BPK15]. This paper derives a set of guidelines for using HTM efficiently, applies the guidelines to develop the concurrent map BT-trees, and evaluates BT-trees.

- Lars Bonnichsen, Artur Podobas: "Using Transactional Memory to Avoid Blocking in OpenMP Synchronization Directives — Don't Wait, Speculate!" in the International Workshop on OpenMP (*IWOMP*) 2015, Springer [BP15]. This paper illustrates our lock and barrier implementations, and evaluates them through an OpenMP library, on a set of benchmarks and microbenchmarks.

1.2 Synopsis

The rest of this dissertation is organized into the following 6 chapters:

Chapter 2 reviews the theory and practice of modern computers power consumption and capabilities, and relates it to our adaptive framework for software development.

Chapter 3 presents a new power governor, `pappeadapt`, which improves energy efficiency by trading processor speed for lower power consumption on workloads where the processor speed is not a performance bottleneck. `pappeadapt` uses a variation of Amdahl's law to predict the optimal tradeoff between speed and power consumption. The predictions are based on performance counter measurements from online experimentation.

Chapter 4 presents speculative techniques which exploit HTM — allowing for further parallelism. The techniques can reduce execution time and energy consumption significantly, but may increase power consumption. We implement the speculative techniques in BLYSK, an OpenMP runtime, replacing its traditional blocking synchronization, with speculative synchronization. Speculation is not always beneficial, so we include mechanisms which minimize the drawbacks of speculation, such as the lemming effect [DLM+09]. The content of this chapter is based on a published paper [BP15].

Chapter 5 explores how to further reduce execution time by designing and manually rewriting code for HTM. We present five guidelines for optimizing code for speculative execution, and illustrate them on a new concurrent

ordered map, BT-trees. BT-trees benefit significantly more from speculation than traditional ordered maps — achieving high time and energy efficiency, but they are still quite straightforward, unlike traditional concurrent ordered maps. Using the the lessons learned from BT-trees, we present to a lock-free map design, ELB-trees, which can be used on computers without HTM support. The lock-free design appears to scale as well as the transactional design, but it is not as general or efficient, because it does not have the strong ordering guarantees provided by TM. The content of this chapter is based on two published papers [BPK15, BKP13].

Chapter 6 concludes the thesis by summarizing our findings, discussing their implications, and outlining possibilities for future work.

Chapter 2

Background

In this chapter we first introduce the terminology for expressing computational performance in Section 2.1, and before the background for reasoning about computational performance in Section 2.2. Finally we describe how we adaptively improve computational performance in Section 2.3.1

2.1 Power consumption

We consume energy for most activities: transportation (burning gasoline), physical activity (burning glucose), and computation (consuming electrical charge). The consumed energy does not disappear, it is transformed into other forms of energy, mainly heat: Consuming a lot of energy, typically produces a lot of heat. We generally want computations to consume less energy, so we produce less heat, reduce the computation's electrical cost, improve the computer's battery life, and potentially allow the computation to finish faster.

The energy required for an activity can be calculated from its power consumption (P) and its execution time (D, short for delay):

$$E = PD$$

We can evaluate computational performance both in terms of execution time (D) and energy consumption (E). Energy is a larger concern today due to stricter requirements on battery life, heat emissions, and electricity bills in modern systems such as mobile devices, low noise personal computers, and large scale data centers. Simply measuring either energy consumption or execution time is fine as longs the other does not change. Traditionally software developers have preferred evaluating performance with execution time, because it is easy measure and most computers we use have a fixed upper bound on power consumption, making D a good approximation of E. However, given modern hardware trends, we cannot generally assume that our power consumption is fixed, rather we should whenever possible measure both energy consumption and execution time when evaluating computational performance.

We can evaluate the combined energy and execution time cost with ED, ED^2, or another similar function. The cost ED puts equal emphasis on energy consumption and execution time, valuating 10 % execution time reduction, as equally valuable as a 10 % energy reduction. The cost ED^2 puts additional emphasis on execution time, valuating a 10 % execution time reduction, as equally valuable as a $1 - (1 - 0.1)^2 = 19\,\%$ energy reduction. The cost functions can be generalized to ED^N, allowing us express the relative worth of time and energy savings.

Unlike software development, hardware development has always been very energy aware: Consuming too much power heats up the hardware, increasing its failure rate making high power consumption prohibitive. A processors power consumption depends on its CPU frequency, or more accurately on its transistors' switching frequency (f). Transistor switching frequencies and voltages are normally scaled together, traditionally satisfying the formula:

$$P \propto f^3 \tag{A}$$

Many hardware developers traditionally used ED^2 as their performance metric, because it is independent of the transistor frequency when (1) Formula (A) holds and (2) the transistor frequency is inversely proportional to execution time ($D \propto f^{-1}$) [MNP02]. Formula (A) is a good rule of thumb for the majority of CMOS transistors deployed from 1980 to 2004. Formula (A) does not hold for entire computer systems, due to the power consumed by other components, such as monitors and storage. Formula (A) does not hold for newer transistors, due to static power consumption being a more dominant factor in their power consumption [HSN04]. The static power consumption is caused by electric charged leaked by the transistors, at a rate which is independent of the frequency, but dependent on the voltage. Dynamic power consumption and CMOS short circuiting represent the other components of transistor power consumption, components which do depend on the frequency ($P(f) = P_{short(f)} + P_{dynamic(f)} + P_{static}$).

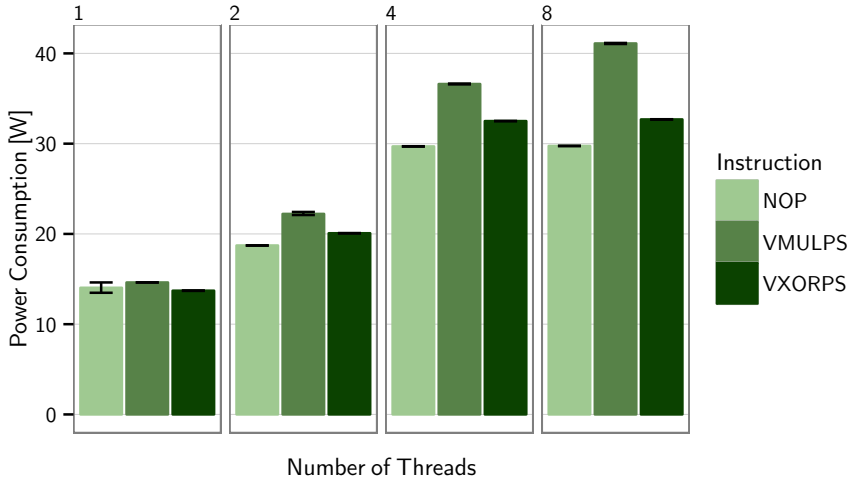


Figure 2.1: Mean processor power consumption as a function of instructions and number of threads. The error bars indicate 95 % confidence intervals computed through bootstrapping.

In other words, the conditions do not hold, so we cannot assume that ED^2 is independent of the frequency.

A processors power consumption is not just a function of its processor voltage and frequency, but also the code it is running and especially the amount of parallelism. For instance, Figure 2.1 shows the processor power consumption of the machine listed in Table 2.1 operating at 3.6 GHz while executing NOP instructions — doing nothing — vectorized XOR instructions — repeatedly inverting 256 bits — and vectorized multiplications. We would expect NOP instructions to be cheaper than vectorized multiplications and bit flips, an expectation which is especially true at high levels of parallelism. In the single threaded case, the

Processor	Intel Xeon E3-1276 v3
CPU Frequency	3.6 GHz, TurboBoost to 3.6 GHz
Processor cores	4 cores, 8 threads
Processor caches	32 KiB L1D, 8 MiB L3
C/C++ Compiler	GCC 5.2.0
Operating system	Ubuntu Server 14.04.1 LTS
Kernel	Linux 3.17.0-031700-generic
C library	eglibc 2.19

Table 2.1: Experimental machine

power consumption is fairly similar for all the instructions, while there is a larger difference in the 8 threaded case. The difference is larger in the parallel case for two reasons: (1) the higher CPU activity leads to a higher fraction of dynamic power consumption and (2) the multiplications yield higher power consumption at 8 threads than 4 due to underutilized ports at 4 threads (See 2.2.2.3), while the NOP and XOR fully utilize their ports at 4 threads.

Modern Intel processors have a feature called TurboBoost, which allows the processor to automatically — at a hardware level — increase the CPU frequency and voltage when the power consumption is below safe thresholds. Modern AMD processors have a similar feature called Turbo Core. These "Turbo" features only increase the CPU frequency, when the processor power consumption is significantly lower than their Thermal Design Power (TDP) — i.e. when the processor can safely increase its power consumption [GBCH01]. Dynamically increasing the CPU frequency has varying effects based on the computers workload and number of instructions executed, as illustrated by Figure 2.2. Enabling TurboBoost, as indicated by going from 3.6 to 4 GHz, does not increase the processors frequency or clock frequency when executing instructions with high power consumption, such as the vectorized multiplication. The NOP and XOR instructions yield higher power consumption than the vectorized multiplications in the single and two threaded cases, because the TurboBoost feature operates at the highest frequency, while the effective CPU frequency is lower in the four and eight threaded cases.

Computer hardware optimizes for power consumption adaptively — based on what it is computing. To reason about software performance and power consumption – and to adaptively optimize software performance – we need an understanding of how the hardware behaves.

2.2 Performant computers

Modern computers apply a lot of hardware cleverness to improve their performance, both in terms of energy and execution time. This section aims to build an intuition about computational performance by briefly reviewing (1) what programs are from a low level perspective, (2) how a modern processor operates, and (3) how to quantify a processor's performance.

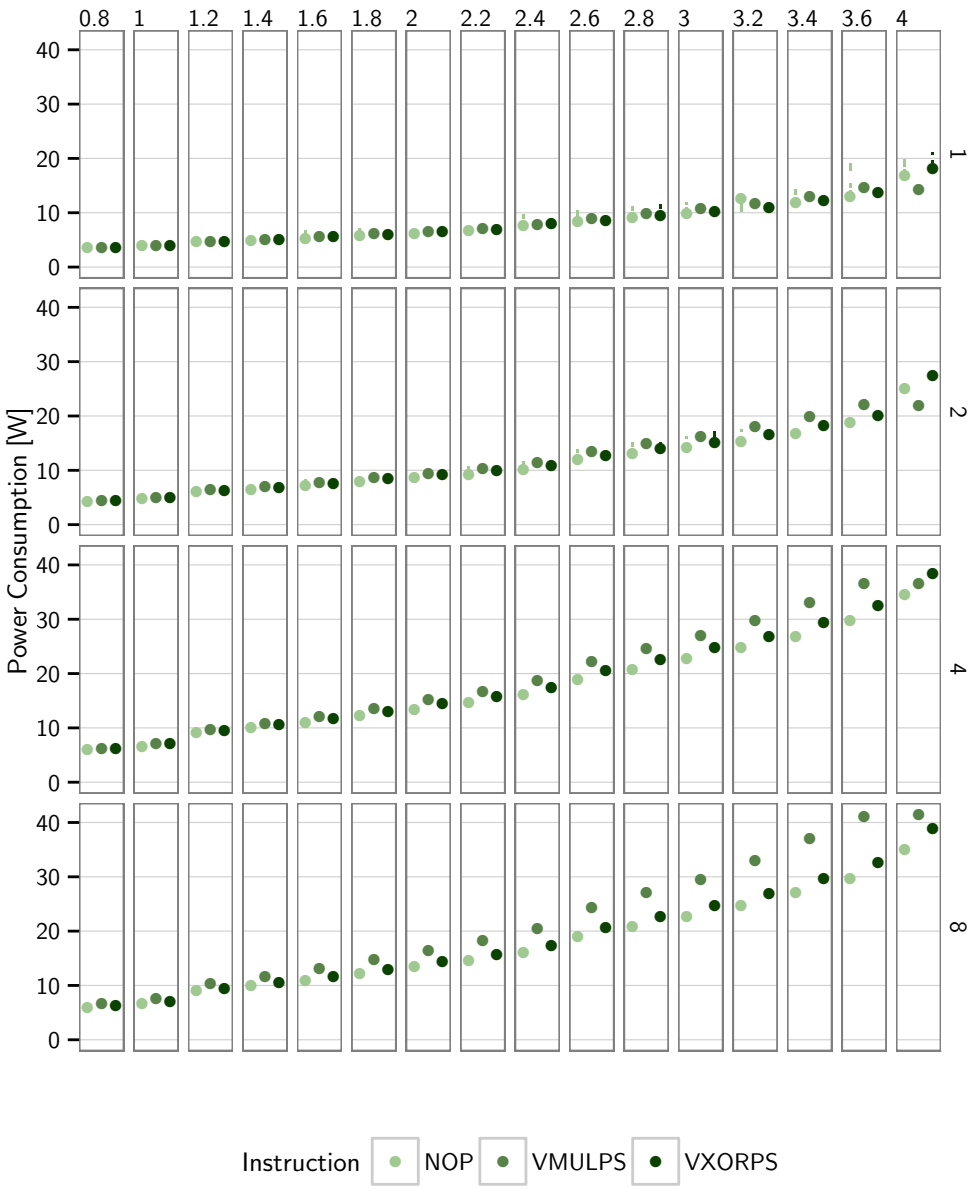


Figure 2.2: Median processor power consumption as a function of instructions (color), number of threads (y-axis), and CPU frequency (x-axis). The small dots indicate individual observations.

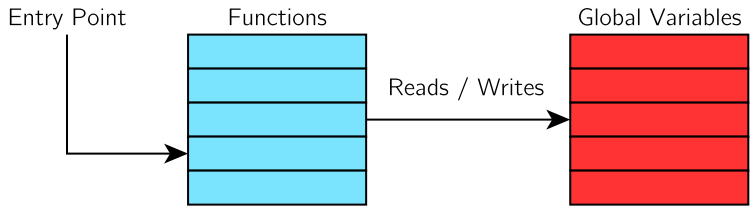


Figure 2.3: From a low level perspective programs consist of functions, global variables, and an entry point.

2.2.1 Instructions: The smallest program fragments

This section provides a brief introduction to what computer instructions are, such that we can discuss program behaviors at a detailed level.

Processors execute programs. Programs consist of three things, as illustrated by Figure 2.3: (1) a set of functions, (2) a set of global variables, and (3) an entry point into the program, indicating where the program starts. The global variables are essentially preallocated storage, while the functions describe what the program can do. The functions are built from instructions, which are the smallest unit of code available. Each instruction represents a small calculation or operation, such as "add A and B", "copy the value of A to B", "divide A and B", "AND the bits in A and B" (bitwise and), "count the number of set bits in A" (population count)", etc.

Instructions are seemingly executed sequentially, one at a time. Processors can change the flow of executed instructions with *jumps* and *branches*: Jumps, as in "jump to A", ensure that the instruction executed next is "A", and branches, as in "if A > B jump to C", conditionally changes the instruction executed next. Jumps and branches can be used to express conditional expressions, loops, and recursion.

Instructions can operate on constants (immediates) and two kinds of variables (1) directly accessed *registers* and (2) indirectly accessed *memory locations*. Figure 2.4 outlines the main differences between registers and memory locations. Memory locations can be addressed as arrays in instructions, for instance `A[B]`, or `A[(B << C) + D]`, allowing memory locations to indirectly refer to different variables depending on the values the other variables. Registers are faster to access than memory locations, but also more limited: they cannot be addressed as an array, and are in short supply, with most processors having 14, 31, or 63 generally usable registers, while memory locations are far more abundant — most processors having memory address spaces of 2^{32} or 2^{64} bytes.

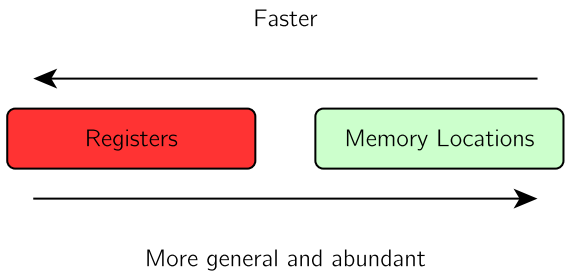


Figure 2.4: Registers are fast but scarce. Memory locations are abundant but slow.

Each processor supports a fixed set of instructions, called its Instruction Set Architecture (ISA). The ISA does not only dictate which operations are directly supported as instructions, but also how memory locations can be addressed, and which combinations of memory locations and registers can be used for instructions. The limited set of instructions and the limited number of registers available, make it difficult and time consuming to write larger programs directly with instructions.

High level languages reduce the work required by programmers with compilers which transform high level code into instructions; compilers which automatically allocate registers for intermediate computations, and select instructions and addressing modes which match the computation and the ISA.

Knowing how to read and write functions with instructions is useful to get a mental model of what a computer does, as well as a means for understanding compiler output. From a processor's ISA you can estimate how many instructions and memory accesses a computation should take; If the compiler output uses more instructions than you expected, then you can either solve the problem by writing the instructions manually, or giving the compiler more information (See Section 2.3 step 3).

2.2.2 Processor cleverness

This section gives a brief overview of some of the ways that modern computer architectures accelerate execution of instructions, providing a background for reasoning about program performance.

Inspecting the instructions of a computation only vaguely hints at the computations execution time, because modern computers use a wealth of tricks to

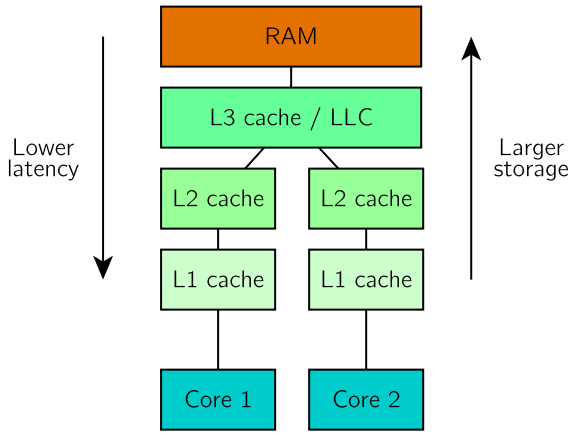


Figure 2.5: A typical cache hierarchy with 3 layers.

speed up common computations. Processors can execute instructions out of order and in parallel, as long as the instructions are sufficiently independent that the changed ordering does not change the outcome of the computation. In the following paragraphs we will describe five such techniques: caching, prefetching, pipelining. register renaming, and superscalar processing.

2.2.2.1 Caches

Caches provide quick — several orders of magnitude faster — access to cached memory locations. Memory accesses which are in a cache (*cache hits*) have lower latency, while memory accesses which are not in a cache (*cache misses*) suffer the full memory latency. Currently most processors provide 2 – 3 layers of caches, as illustrated by Figure 2.5, which dramatically reduce the average memory access latency, and allow memory operations to proceed in parallel.

The cache layers range from small and fast, to large and slow, for instance the Intel Haswell i7-4770 has an L1 cache latency of 4-5 cycles, L2 cache latency of 12 cycles, L3 cache latency of 36 cycles, and a typical RAM latency of 200 cycles [7-c]. In most applications the majority of memory accesses are to locations cached in in the L1 cache, which significantly speeds up the average memory latency.

Each cache layer tracks a fixed number of cache lines, which are typically 32, 64, or 128 continuous bytes. The cache lines are automatically added to the cache whenever they are used. Because caches have fixed sizes, adding a new cache line also evicts an old cache line, but in typical use the most recently accessed

data will be cached. Memory access through caches typically speed up access to recently accessed data (*temporal locality*) — because that memory most still in the cache, i.e. not evicted yet — and data adjacent to recently accessed data (*spatial locality*) — because adjacent data is likely in the same cache line.

Caches are separate components in the processor, responsible for accessing memory. All of the cache layers can also keep multiple memory access in flight at any time, providing a high memory bandwidth in spite of the memory latency. The lowest layer of the caches are usually duplicated, providing a private L1 cache for each processor core, while higher cache layers are shared between cores, in order to use the caches efficiently, as illustrated in Figure 2.5.

The highest layer of the cache, or the Last Level Cache (LLC) as it is normally called, is responsible for fetching memory from RAM. LLC's typically fetch multiple cache lines at a time, for instance two consecutive cache lines, because the program is likely to need the memory — and fetching two consecutive cache lines from RAM concurrently takes significantly less time than fetching two cache, one at a time — further strengthening the benefit of spatial locality.

Most modern processors also include *prefetchers*, which further exploit spatial locality, by predicting which memory locations are likely to be used next, and fetching the memory before any CPU requests it. Prefetchers can dramatically reduce memory access latency when there is a delay between each consecutive memory access, and the spatial distance between consecutive memory accesses is predictable. For instance, a program which processes consecutive cache lines at a rate of one line per 1000 clock cycle will likely to benefit from a prefetcher, because the prefetcher is likely to predict the memory access pattern, and fetch the memory before it is used for the first time.

2.2.2.2 Pipelines

The processor can execute a series of instructions in parallel by executing separate stages of the instructions in a pipeline. Pipelining splits instruction execution into several shorter stages, which allows the processor to operate at a lower voltage — or a higher frequency. Instruction execution is often separated into several stages — for instance stages for fetching instructions, decoding instructions, performing calculations, etc.— with typical modern processors having10-20 pipeline stages. Pipelining works really well as long as the processor knows which instruction is executed next, but it has a harder time handling branches and indirect jumps.

A traditional pipelined processor has one instruction in flight for every pipeline

stage. When a pipelined processor encounters a branch, it has to guess whether the branch will be taken or not, because the instructions which determine the outcome of branch may be in flight. The pipeline continues operating as usual if the processor guesses correctly, but if the processor mispredicts the branch, it has to stop the instructions currently in the pipeline, waiting for them to finish. Mispredicting a branch means that the pipeline has to be flushed, i.e. a typical cost of 10-20 cycles. Modern processors have branch prediction hardware, which track the recent history of the most frequently encountered branches, to reduce branch misprediction rates. Branch prediction hardware tend to be accurate when the branches are easy to predict — for instance if they are mostly taken, or taken every other time they are executed — and inaccurate when the branches are seemingly random.

2.2.2.3 Out of order processing

Out of order processors apply register renaming and superscalar execution, to exploit implicit parallelism and minimize latencies.

With register renaming, the processor does not immediately write to registers, but to a temporarily allocated buffer. This scheme allows the processor to execute instructions in parallel with instructions that use the old register value, as well as instructions which use the new value. Register renaming allows the processor to separate the start of an instructions execution (instruction issue), from the end of an instructions execution (instruction retire), allowing instructions to be underway (in flight) for a long time. Having multiple instructions in flight can hide the latency of memory accesses or slow instructions, by executing other instructions concurrently, as well as permitting further parallelism such as superscalar execution.

With superscalar execution, the processor has multiple functional units, also known as ports, each of which can execute an in flight instruction. For instance, Intel Haswell processors have 8 ports per core, each of which can execute a subset the instructions in the ISA. The ports are rarely fully utilized, so modern processors use simultaneous multi processing (SMT, or hyperthreading), letting multiple CPUs operate on a single set of ports [TEL95]. Having multiple CPUs allows the processor to execute multiple threads of the program in parallel, but the CPUs which share ports will individually execute slower if the ports become contended.

Modern out of order processors can issue several instructions per cycle — as long as the instructions are aligned properly — and retire several instructions per cycle, exposing a great deal of parallelism implicitly. Unfortunately, CPUs

occasionally have to stop issuing new instructions — an action known as a *frontend stall* — due to two limitations in register renaming: (1) you can only have a limited number of instructions in flight — due to the limited number of temporarily allocated buffers — and (2) you can only issue a limited number of instructions after the oldest in flight instruction — because you need to track dependencies between instructions.

2.2.2.4 Alternate processor designs

When combined, caches, pipelines, branch prediction, register renaming, and superscalar execution enable implicitly parallelizing instruction and memory access. The hardware required all of the cleverness takes requires a lot of transistors. Alternative processor designs, such as graphics processors (GPUs), deemphasize these forms of implicit parallelism, allowing them to fit more cores and wider vector units in the same space and power budget. Alternative processor designs tend to favor the energy efficiency through parallelism in applications which have very regular memory access patterns. As a consequence, most systems use GPUs as a way to complement regular processors, rather than replace regular processors: When GPUs can execute programs efficiently — when the program is embarrassingly parallel and has a regular memory access patters — GPUs are often significantly more efficient than regular processors, but graphics processors are not always able to execute programs efficiently. Despite the potential savings from applying GPUs, we believe the architecture of regular processors will continue to be viable for the foreseeable future.

2.2.3 Performance counters

Modern processors include *performance counters* which count specific events in the processor, such as:

- Executed instructions (retired instructions), which can be further classified as branches, loads from memory, stores to memory, and computations.
- Mispredicted branches and CPU frontend stalls.
- Cache misses and cache hits at the various layers of the cache.

These performance counters can guide optimization efforts:

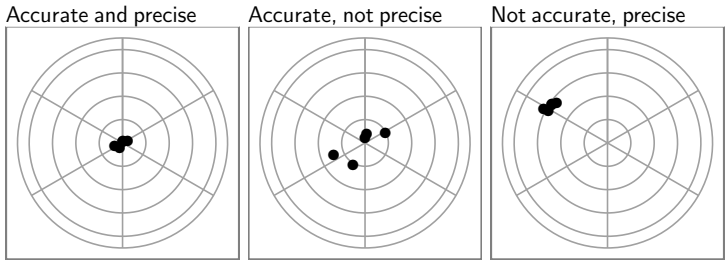


Figure 2.6: Precise measurements have low variation. Accurate measurements have little bias.

- If the workload causes many cache misses, then try to minimize them, by reordering the data or the order of the memory accesses.
- If the workload executes many instructions, then look for a more efficient algorithm or implementation, which will execute fewer instructions.
- If the workload causes many branches, or branch mispredictions, then try to replace the branches with arithmetic, or make the branches easier to predict.

The operating system Linux has a profiling tool `perf`, which uses the Linux system call `perf_event_open` to access performance counters. `perf` can sample the performance counters while applications are running to determine which parts of the application causes most of the events. The performance counters are managed by the operating system because they are a fixed hardware resource, which applications must share. Most modern operating systems have similar tools and system calls to Linux's `perf` and `perf_event_open`.

Linux also provides APIs (sysfs) for reading the power controllers on modern computers, such as those on Xilinx Zynq boards [Sri15] and Intel processors (Intel RAPL) [Int15a]. The measurement facilities for performance counters and power consumption are not perfectly accurate due to hardware and software limitations, but they are representative enough to identify and classify bottlenecks, making it easy to direct optimization efforts.

The fundamental difference between intrusive and non-intrusive profilers is that intrusive profilers are precise, while non-intrusive profilers are accurate, as illustrated by Figure 2.6. Intrusive profilers change the application under test, to precisely detect whether there is anything wrong with the application. Intrusive profiler output is precise — repeating the experiments will yield the same

results — but it is not accurate — the intrusive profilers change the application, so it can be misrepresentative. Non-intrusive profilers accurately detect how well the implementation performs: Their output is accurate because the profiler does not change the function under test, but it is not precise because of measurement errors. Measurement errors are caused by inaccurate performance counters and external events, such as the room temperature, background processes running on the computer, iterrupts, randomized memory layout, and the operating systems state [WM+08, WTM13]. We can usually compensate for poor measurement precision by repeating experiments, so we can look at the distribution and median of the results, but we cannot compensate for poor measurement accuracy.

2.3 Optimizing software performance

In this thesis, we adapt systems to improve performance by either: (1) lowering the voltage and frequency level, increasing execution time, but lowering power consumption, or (2) running more efficiently, reducing execution time. We take the first approach in Chapter 3, and the second approach — exposing further parallelism — in Chapter 4 and 5. Section 2.3.1 describes our adaptive procedure for improving software performance, without exposing further parallelism.

2.3.1 Adapting software for performance

We optimize functions through an adaptive process — alternating between measuring, reasoning and changing the function — in order to gain confidence in the implementations correctness, ensure that it is efficiently compiled, and meassure its performance. The procedure typically involves iterating over 5 steps:

1. Write an implementation of the function.
2. Test the implementation using:
   (a) Static analysis tools (Frama-C, lint, etc.) and compiler options (See Listing 2.1) to detect likely bugs.
   (b) Tests that give random and valid input to function, while checking that the function output is correct. Such tests follow a Hoare Logic precondition / postcondition style, and are known as stress tests, fuzzing, and quick checking.

   (c) Intrusive profilers (such as `valgrind`, `gprof`, and `pin`) to detect memory leaks and incorrect use of APIs.

3. Sanity check the implementation with an intrusive profiler, to ensure that code is compiled efficiently.

4. Use a non-intrusive profiler (such as `perf` and oprofile) to detect and minimize bottlenecks.

5. Go to step 1 if the implementation is not fast or energy efficient enough, otherwise accept the implementation.

The following paragraphs illustrate step 2 to 4, showing the tools and methods we use in each step.

Step 2.a: Testing with static analysis tools
Static analysis tools and compilers with additional warnings enabled can help detect programming errors. These tools are especially useful in languages such as C and C++ which operate under the motto "the programmer is always right" letting programmers do strange things such as type casting, using unitialized variables, and omitting return values in functions. Allowing the programmer to do strange things often leads to errors; errors which compilers and static analysis tools can detect.

Modern compilers, such as GCC, LLVM, and ICC, support a wealth of *warning options* for detecting likely bugs, disallowing language features, and suggesting improvements for the code. These compilers generally do not enable all of their warning options by default, or even when using options such as `-Wall`, is often understood to mean "enable all warnings", but actually means "enable all warnings that most people care about": We are not "most people", in fact very few people have the exact needs of "most people", and could benefit from more aggressive warnings from compilers and static analysis tools. To illustrate, we generally use the warning options in Listing 2.1 when compiling with GCC, a total of 23 for detecting issues such ranging from excessive stack usage, and dangerous type casts, to memory accesses outsde array bounds. In our opinion, using static analysis and compiler options to detecting likely bugs, and dangerous language constructs is important for all code, not just performance critical code, but performance critical code may further benefit from compiler based suggestions for optimizations.

Step 2.b: Testing with fuzzing
Fuzz tests are small programs which generate valid input for functions, call the

```
gcc -c file.c -o file.o -O2 \
 -W -Wpedantic -Wextra -Wall -Wno-unused-function -Wabi \
 -Wstrict-aliasing=2 -Wstrict-overflow=1 -Wabi -Wcast-align \
 -Wstack-usage=32768 -Wframe-larger-than=32768 -Wcast-qual \
 -Wsync-nand -Wtrampolines -Wsign-compare -Wformat-signedness \
 -Werror=float-equal -Werror=missing-braces -Werror=init-self \
 -Werror=logical-op -Werror=write-strings -Werror=nonnull \
 -Werror=array-bounds -Werror=char-subscripts -Werror=address \
 -Werror=enum-compare -Werror=implicit-int -Werror=main \
 -Werror=aggressive-loop-optimizations -Werror=sequence-point \
 -Werror=parentheses -Werror=pointer-sign -Werror=return-type \
 -Werror=uninitialized -Wno-error=maybe-uninitialized \
 -Werror=volatile-register-var -Werror=ignored-qualifiers \
 -Werror=missing-parameter-type -Werror=old-style-declaration \
 -Wsuggest-final-types -Wsuggest-final-methods \
 -Wsuggest-attribute=format -Wsuggest-attribute=pure \
 -Wsuggest-attribute=const -Wsuggest-attribute=noreturn \
 -Wsuggest-override
```

Listing 2.1: How we typically use the GCC compiler to detect likely errors (bright highlights), forbid certain errors (medium highlights), and suggest function annotations (dark highlights).

functions with the input, and check that the functions produce valid outputs. We usually check the output by comparing it to the output of known good functions, or by checking the function satisfies its postconditions.

Listing 2.2 illustrates fuzz testing for a logarithm base 10 function over the integers from 1 to 99. The fuzz test (Line 13) tests the our function (Line 3) on randomly generated inputs from 1 to 100 (Line 16) by comparing with the output from a known good implementation (Line 19). Because the function is defined over a small domain of inputs, we can use a simple implementation consisting of two branches, where we expect most inputs only execute one branch, since most values are presumably greater than 10. Having a simple implementation often leads to faster code which the compiler can optimize more efficiently, In other words, being aware of a functions input domain or preconditions is often important for implementing functions efficiently.

Fuzz tests are trivial to write when you know the functions preconditions and postconditions. In our example, we have a very strict precondition – the input must be in the range $[1; 99]$ small input domain, so we could also test the implementation exhaustively — over all valid inputs — but exhaustive tests are not always feasible. Randomized tests can also be used to evaluate a function implementations performance, later in the adaptive development procedure. We can usually generate representative random input much faster than the functions being tested execute, by using state of the art pseudo random number genera-

```
/** A log10 function over the
    domain 0-99. */
int log10New(int val) {
  if(val > 10)
    return 2;
  if(val > 1)
    return 1;
  return 0;
}

/** Compare log10New's output
  with log10 over random input */
void log10Test() {
  int i = 1000000;
  while(--i) {
    int in = random(1, 99);
    int r1 = log10New(in);
    int r2 = log10(in);
    assert(r1 == r2);
  }
}
```

Listing 2.2: Fuzz testing a logarith base 10 function

tors, such as Mersenne Twister [MN98] or lfib4 [Mar99]. Since fuzz testing can improve our confidence in a function implementations correctness and performance, we recommend fuzz testing any safety or performance critical functions — fuzz testing may be overkill for trivial and non-critical code.

Step 2.c: Testing with intrusive profilers
We test whether uses APIs correctly with intrusive profilers, such as `valgrind`. `valgrind` can count the number of times functions are called — indicating whether they are used correctly — detect misaligned data access — indicating invalid type casts, and memory leaks. These tests are useful both for safety and performance critical code, because mistakes such as memory leaks often harm both correctness and performance. The following command shows how to use `valgrind` to detect misuses of APIs, in the program `linear_search`:

```
valgrind -v --fair-sched=yes --track-origins=yes \
  --leak-check=full --show-leak-kinds=all "./linear_search"
```

Step 3: Sanity checking with intrusive profilers
After checking the correctness of the function implementation, we can use intrusive profilers to detect whether the implementation was compiled efficiently. We need to check the quality of compiled performance critical functions, because the compiler will sometimes produce inefficient code. Inefficiently compiled code is usually caused by abstractions made in the program, abstractions which we assume the compiler can optize. There are limits to how much a compiler can optimize our programs automatically, because compilers are tools; not magic wands. We can check the quality of compiled code with with intrusive profilers, such as `valgrind` and `pin`, by asking them to count the number of instructions executed or memory references. Poorly compiled code will often execute far more instructions and memory references than neccessary, making them easy to

```
1 // Running valgrind from a shell
2 valgrind --fair-sched=yes --tool=cachegrind --cache-sim=yes \
3   "./linear\_search"
4 cg\_annotate --show='Dw' \$(ls -t cachegrind.out.* | head -n 1) \
5   --auto=yes --threshold=2
6 cg\_annotate --show='Ir' \$(ls -t cachegrind.out.* | head -n 1) \
7   --auto=yes --threshold=2
8
9 // Valgrind output showing the problem
10     Dw
11           . void linearSearch(int*i, int*array, int value, int n) {
12 33,000,000   for(*i = 0; *i < n; *i += 1)
13           0  if(array[*i] == value)
14           .     return;
15           . }
```

Listing 2.3: Using `valgrind` to detect code paths which contribute to more than 2 % of the total number of memory writes and instructions executed in the program `linear_search`.

detect.

Listing 2.3 illustrates how to use `valgrind` to look for poorly compiled code in `linear_search`, an exaple program performing 1,000,000 linear searches over an array of length 32. After running the code (Line 2) we ask valgrind to show us the code which causes most writes (Line 4), and we are informed that the linear search performs 33,000,000 writes (Line 12), whereas we would expect it to perform 1,000,000 writes — one write per call to the function. The compiler generated code which writes to memory whenever the code increments the index `i`, because the compiler did not know whether `i` overlaps with the array — a problem known as pointer aliasing, because it occurs when two pointers point to the same data. Since the function `linear_search` does not indicate whether the pointer alias the compiler has to assume that they can alias, generating code which writes to memory whenever `i` is updated, rather than just at the end of the function call. Pointer aliasing is a common problem, and in this case we could solve the problem in three ways: (1) tell the compiler that `array` and `i` do not overlap, by annotating them with the type qualifier `restrict`, (2) move the function `linearSearch` to a header file, so it can be inlined from its call sites and the compiler can determine if the `array` and `i` alias, or (3) ask the compiler to perform whole program optimization or link time optimization, so the compiler can determine if the `array` and `i` alias.

Intrusive profilers can also reveal problems such as redundant branches, which were not optimized out because the compiler did not know the valid range of variables. Such problems can be fixed by restricting the valid range of variables, for

instance: `if(A < 1) __builtin_unreachable();` will inform compilers such as GCC and LLVM that `A` cannot be smaller than 1.

Using intrusive profiler output often requires that we reason about performance critical code using a mental model of what the code's instructions should be. The output makes it easy to filter away non performance critical code, saving a lot of developer effort compared to manually inspecting the code's instructions, but it is still time consuming and it only makes sense for applications which are performance critical.

Step 4: Detecting bottlenecks with non-intrusive profilers
After whether the function implementation is correct and properly compiled, we can use non-intrusive profilers such as `perf` to measure the performance of the function, and to detect its bottlenecks. To illustrate, we can use `perf` to measure the performance of a program, such as `linear_search`, with the shell command:

```
perf stat -e L1-dcache-load-misses -e L1-dcache-store-misses \
  -e cache-misses -e cycles -e instructions -e cpu-clock \
  -e task-clock -e page-faults -e minor-faults "./linear_search"
```

The measurement can be used to compare the performance metrics of different implemntations, allowing us to shed some light on how they perform. We usually supplement the performance metrics with measurements of energy consumption through APIs like Intel RAPL.

`perf` to detect bottlenecks in programs — in this case `linear_search` — using commands like:

```
perf record --call-graph dwarf -F 39 -e task-clock "./linear\_search"
perf report
```

The bottlenecks indicate where the program spends its time, allowing us to direct further optimization efforts.

Discussion
Executing an iteration of the adaptive procedure, from step 1 to 4, gives us confidence that the function implementation is correct, efficiently compiled, and tells us how well it performs. The adaptive procedure contains nested adaptive behavior; we repeatedly adapt the implementation in step 2.a, 2.b, 2.c, 3, and 4. We change the function implementation and measure its behavior until it

satisfies the requirements that we expect from our reasoning. The process is time consuming, but we have found it worthwhile, because it gives us a framework for supporting development of performance and safety critical code.

The format that we have described only operates on a single function. The adaptive process is normally nested in an adaptive process for optimizing all performance critical functions in the whole program. To minimize the effort required for implementing functions, we normally only use the development procedure for performance and safety critical code. While the process requires a good deal of effort, we have also outlined cases where the different steps can be skipped, and we believe that it is the minimum required effort to be reasonably confident that our implementations are efficient and correct.

We have used the development procedure while developing the middleware described in the remainder of this thesis. In some cases, we use the adaptive procedure was mostly to ensure correctness of the code, in other cases we use the procedure to optimize the code. We have generally found that the the effort required for the adaptive development procedure is lower than normal, unstructured, and ad-hoc program development. The main benefit from the procedure came in the form of improved implementations, and understanding of errors in early forms of the middleware, but in some places we also improved our designs by repeatedly changing, measuring, and reasoning about it.

Chapter 3

Workload Aware Energy Management

In Chapter 2 we showed that setting CPU frequencies is a delicate tradeoff between execution time and power consumption. Reducing a processor's frequency and voltage trades a reduced its power consumption, for an increased execution time: Reducing the processors dynamic power consumption is not always a viable method for reducing the systems energy consumption due to the static power consumption (current leakage) and the power consumption from other components, such as memories and screens.

In this chapter we introduce `pappeadapt` (pronounced papp-e-adapt), a power governor which automatically reduces the CPU frequency when reducing the frequency does not significantly increase the execution time. `pappeadapt` reduces the processors geometric mean energy consumption by 41.5 %, ED by 16.4 %, and ED^2 by 4.7 %, on the benchmark suites NPB 3.3.1 [BBB+91], SPEC2006 [Hen06], PARSEC 3.0 [Bie11], SPLASH-2x [WOT+95].

`pappeadapt` uses a combination of online experimentation and prediction to adapt the CPU frequency. The predictions are based on Amdahl's law, cheap to calculate, and very precise.

This chapter introduces:

- Two microbenchmarks which exemplify compute-bound and memory-bound workloads.
  - The compute-bound microbenchmark can be used to identify good fixed CPU frequencies which provide good energy costs (ED^N).
- The proposal and evaluation of a model, based on Amdahl's law, for predicting the execution time of memory-bound workloads.
- An adaptive method for optimizing CPU frequencies to the systems current workload.

This chapter is structured as follows: Section 3.1 illustrates why, when, and how we can save energy by reducing CPU frequencies. Section 3.2 evaluates the effects of statically or adaptive reducing CPU frequencies. Section 3.3 relates our work to prior work, Section 3.4 suggests improvements to `pappeadapt` based the evaluation and the related work, and finally Section 3.5 summarizes our findings.

3.1 CPU frequencies and workloads

We can reduce energy costs (E, ED, and ED^2) by selecting the right CPU frequency. The optimal CPU frequency is not necessarily the highest or lowest supported CPU frequency, and the optimal frequency also depends on the system's workload. To illustrate this point Figure 3.1 shows the processor energy consumption as a function of the CPU frequency in a memory-bound and a compute-bound microbenchmark, on an Intel Xeon based server server dubbed `xeon` (see Table 3.1.

The most energy efficient CPU frequencies on `xeon` are $f_{hi} = 2.2\,\text{GHz}$ and $f_{lo} = 1.6\,\text{GHz}$ in the compute and memory-bound workloads respectively. Figure 3.2 shows the relative energy consumption at f_{lo} f_{hi} and the baseline (4 GHz and 3.6 GHz). On `xeon`, restricting the CPU frequency to f_{hi} saves 32 % to 39 % energy, and restricting the CPU frequency to f_{lo} saves 25 % to 45 %. Simply restricting the CPU frequency to f_{hi} or f_{lo} can improve energy efficiency significantly.

We have made similar observations on a small Intel NUC dubbed `snuc` (see Table 3.1), and when optimizing for energy delay products (ED), or energy delay squared products (ED^2): The optimal CPU frequency depends on the workload, it is lower for memory-bound workloads (f_{lo}) and higher for compute-bound workloads (f_{lo}). Optimizing for ED yields higher optimal CPU frequencies than optimizing for E, and optimizing for ED^2 yields the highest optimal CPU

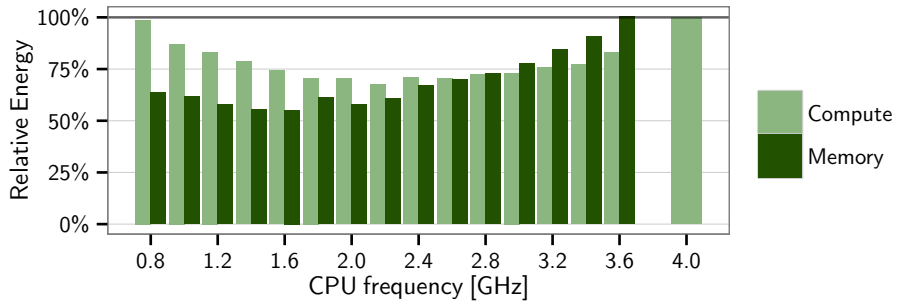


Figure 3.1: `xeon`'s processor relative energy consumption in two workloads. Section 3.1.2 describes the workloads. Lower is better.

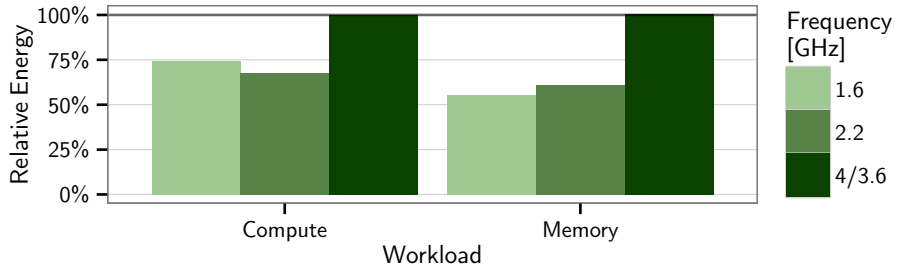


Figure 3.2: Energy consumption at 1.6 GHz, 2.2 GHz, and the baseline. Lower is better.

frequencies. The highest possible CPU frequency (the baseline) frequently yields the optimal (lowest) ED^2 in compute-bound workloads. Restricting the CPU frequency to f_{lo} in the compute-bound workload increases ED by 41 % and ED^2 by 181 % on `xeon`. Simply restricting the CPU frequency to f_{hi} may slightly improve ED and ED^2, while restricting the CPU frequency to f_{lo} may dramatically harm ED and ED^2.

Relationship between throughput and Frequency

Figure 3.3 explains why the optimal CPU frequency is workload dependent: Throughput is workload dependent, growing linearly when compute-bound, and sublinearly when memory-bound, while power consumption grows with the frequency regardless of the workload. Compute-bound workloads maximize their energy efficiency at higher CPU frequencies than memory-bound workloads, because compute-bound workloads' throughput benefit more from high CPU frequencies.

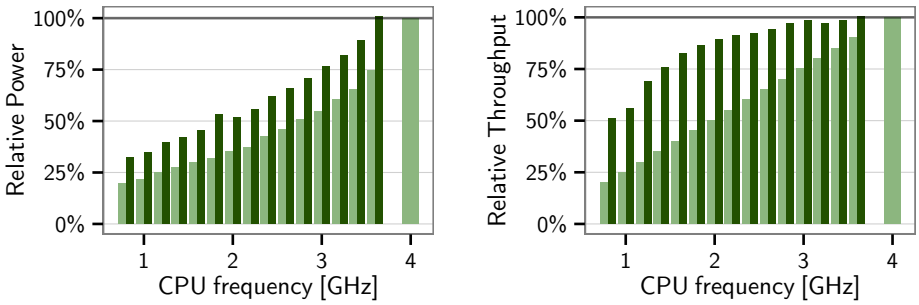


Figure 3.3: `xeon`'s processor power consumption and throughput in two workloads. Lower is better.

Memory-bound workloads spend the majority of their execution time off-chip fetching memory from RAM, while most execution time in compute-bound workloads is spent on-chip, crunching numbers, branching, or fetching memory from caches. On-chip activities speed up linearly with the CPU frequency, while off-chip activities are independent of the CPU frequency, allowing us to model execution time from the time spent on on-chip activities (C_{on}) measured in clock cycles, and the time spent on off-chip activities, measured in seconds:

$$D = D_{off} + \frac{C_{on}}{f}, \tag{1}$$

The model can be derived from Amdahl's law $\left(D(n) = D(1)\left(s + \frac{1-s}{n}\right)\right)$ by substituting the speedup factor (n) with f, and the unimproved fraction (s) by $\frac{D_{off}}{D_{off}+C_{on\,f-1}}$. The model assumes that on-chip and off-chip activities do not overlap, simplifying the model, but it can still accurately predict relative execution time.

Name	`xeon`	`snuc`
Processor	Intel Xeon E3-1276 v3	Intel Celeron N2820
CPU Frequency	3.6 GHz, boost to 4 GHz	2.13 GHz, boost to 2.4 GHz
Processor cores	4 cores, 8 threads	2 cores, 2 threads
Processor caches	32 KiB L1D, 8 MiB L3	??? L1D, 1 MiB L2
C/C++ Compiler	GCC 5.2.0	GCC 5.2.0
Operating system	Ubuntu Server 14.04.1 LTS	Fedora LXDE 21
Kernel	Linux 3.17.0-031700-generic	???
C library	eglibc 2.19	glibc ???

Table 3.1: Experimental machines

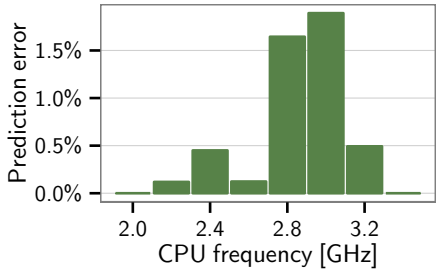


Figure 3.4: Difference between measured and predicted execution time from formulae (2) and (3). Lower is better.

Predicting execution time

Based on two performance measurements at different CPU frequencies, we can predict the performance at any CPU frequency, with the following relation, derived from Equation (1):

$$D \propto m + f^{-1}, \tag{2}$$

where m represents how memory-bound the workload is. Given D_1, the execution time at CPU frequency f_1, and D_2, the execution time at CPU frequency f_2 (with $f_1 \neq f_2$) we have:

$$\frac{D_1}{D_2} = \frac{m + f_1^{-1}}{m + f_2^{-1}} \Leftrightarrow \underline{\underline{m = \frac{D_2 f_2 - D_1 f_1}{f_1 f_2 (D_1 - D_2)}}} \tag{3}$$

We can predict the relative execution time at any CPU frequency with Formula (2) once we know m. Figure 3.4 shows the difference between the predicted and measured execution time in the memory-bound workload, where the predictions are based on the execution times at the CPU frequencies 2.0 GHz and 3.4 GHz. The predictions deviate at most 1.9 % from the measurements, indicating that the predictions are more than accurate enough. We can predict which CPU frequency is most energy efficient from the estimated execution times and power consumption at every CPU frequency.

3.1.1 The power manager

Our power manager `pappeadapt` optimizes energy costs (ED^N) by adapting the CPU frequency to the current workload. We implemented `pappeadapt` as a seperate program, which monitors all applications on the system as they are running (online). `pappeadapt` experimentally uses the CPU frequencies f_{lo} — the lowest potentially optimal CPU frequency — and f_{hi} — the highest potentially optimal CPU frequency — to identify how memory-bound the workload using equation (3). Based on the estimate `pappeadapt` predicts which frequency is optimal with formula (2). `pappeadapt` adapts to changing workloads by repeating the experiment and prediction approximately once per second. In order to make the online adaptivity, `pappeadapt` requires a brief training step.

Ahead of time training
`pappeadapt` requires a training session before use: run the compute and memory-bound microbenchmarks at each possible CPU frequency, while measuring the systems power consumption, cache misses, and execution time. From the training data, we can determine:

f_{lo} The most efficient CPU frequency for the memory-bound workload.

f_{hi} The most efficient CPU frequency for the compute-bound workload.

$\vec{P}$ The relative power consumption at every CPU frequency. We measure this for the compute-bound workload, and assume that proportional power consumption is independent of the workload.

cm_{lo} 10 % of the cache miss rate of the memory-bound workload. We assume workloads with fewer cache misses are compute-bound.

The training step takes less than 2 minutes on `xeon`. We conduct cache measurements with Linux `perf` and power measurements using Intel RAPL, which measures the energy consumed by the entire processor package (cores, caches, integrated GPUs, etc.).

Online adaptivity
`pappeadapt` adapts the systems CPU frequency in real time with the 4 step procedure in Listing 3.1:

1. Measure the performance of the frequency f_{hi} for 0.1 seconds (see Line 3). Repeat this step as long as the cache miss rate (`cm`) is very low (Line 4).

```
1 while(true) {
2   do { // 1. Experiment with fHi
3     hi = measure(fHi, 0.1);
4   } while(hi.cm <= cmLo);
5   // 2. Experiment with fLo
6   lo = measure(fLo, 0.1);
7   // 3. Predict the best frequency
8   if (lo.ir >= hi.ir) {
9     best = F[0];
10  } else {
11    m = (fHi*lo.ir-fLo*hi.ir) /
12      (fHi*fLo*(hi.ir-lo.ir));
13    bestCost = P[0] *
14      pow(m + 1/F[0], N + 1);
15    best = F[0];
16
17    for (i = 1..nF-1) {
18      newCost = P[i] *
19        pow(m + 1/F[i], N + 1);
20      if(newCost < bestCost) {
21        bestCost = newCost;
22        best = F[i];
23      }
24    }
25  }
26  // 4. Use the best frequency
27  for(i=1..8) {
28    tmp = measure(best, 0.1);
29    if(tmp.cm<cmLo && best!=fHi)
30      break;
31  }
32 }
```

Listing 3.1: The `pappeadapt`'s energy saving procedure.

2. Measure the performance of the frequency f_{lo} for 0.1 seconds (Line 6) .
3. Predict the best CPU frequency, in two steps:
   (a) Use the measurements to estimate how memory-bound the workload is with Equation (3) (Line 11 – 12).
   (b) Use the estimate to predict the cost of the different CPU frequencies with Formula (2). Select the frequency with the lowest cost (Line 17 – 24).
4. Use the predicted best CPU frequency for 0.8 seconds (Line 27). Restart the procedure early, if the system exhibits very few cache misses (Line 29).

The procedure uses the function `measure(frequency, time)`, which sets the CPU frequency to `frequency`, measures all applications on the computer for

```
volatile int a = 0;
while(a < 512 << 20)
  a++;
```

(a) Compute-bound microbenchmark

```
char array[256 << 20];
memset(array, 1, 256 << 20);
for(int j = 0; j < 16; j++) {
  for(int i = 0; i < 256 << 20; ) {
    char inc = array[i];
    array[i] = inc * inc;
    i = i + inc * 256;
  }
}
```

(b) Memory-bound microbenchmark

Listing 3.2: The microbenchmarks

`time` seconds, and returns the cache misses per second (`cm`) and instructions retired per second (`ir`). We estimate the workloads performance with the instruction retire rate (`ir`) — the number of instructions executed — implicitly assuming that the instruction retire rate would be constant if we did not change the CPU frequency. Generally we cannot assume that the instruction retire rate will be constant — the workload can change while we were measuring, and it can enter a different phase of computation — but because we are measuring in short durations the workloads behavior is likely unchanged [SEH11]. As such, our predictions are based on Equation (3), by substituting $D = \frac{1}{ir}$.

3.1.2 The microbenchmarks

Listing 3.2 illustrates the two microbenchmarks. Both of the microbenchmarks involve very little work from the operating system, the hot spots in the benchmarks contain no system calls, and they cause very few page faults. We did not investigate the impact of system calls and sleeping processes, because the default Linux power governors already handle such workloads efficiently.

The compute-bound microbenchmark decrements a volatile counter $512 \cdot 1024^2$ times. The benchmarks causes almost no cache misses (INSERT TYPICAL NUMBER), since the counter and the code fit in the processors cache. The compute-bound microbenchmark represents an extreme case, where all activities are on-chip.

The memory-bound microbenchmark iterates over an array of 256 MB bytes 16 times. The code uses the array contents as the stride, ensuring the reads cannot run concurrently, and all array elements are initialized to 1, ensuring a consistent stride of 256 bytes. We unrolled the innermost loop to 16 iterations to minimize branching overheads. We also prevent the compiler from optimizing

the code away by telling it that the arrays contents is replaced by an unknown value after the `memset` (after Line 2), with GCC inline assembly.

The memory-bound workload represents an extreme case, where most activities are off-chip. It is also an extreme case in that all of the data accesses are dependent, preventing any form of memory level parallelism (MLP), which means that each cache miss causes as much harm as possible. Low cache miss rates are good indicators for compute-bound workloads, because workloads with lower cache miss rates than the memory-bound benchmark cannot be memory-bound.

3.1.2.1 Other workloads

For our initial studies we also wrote a microbenchmark which iterates over an array of 256 MB 64 times, reading and writing every 256 bytes, using independent reads. On `xeon` this benchmark has a 9.4 times higher cache miss rate than the memory-bound microbenchmark (on average 153.8 million cache misses per second vs 16.4 million cache misses per second), and better throughput scaling over frequency. The benchmark has a much higher cache miss rate than anything we have observed in real single threaded applications, illustrating that *high cache miss rates are not a good indicators for memory-bound workloads.*

Our initial studies also looked at the scaling of throughput over frequency for SIMD instructions, expensive instructions (division and population count), TLB misses and branch mispredictions, with a corresponding set of microbenchmarks. We found that TLB misses scaled similar to the memory-bound workload when they also cause cache misses, and similar to the compute-bound workload otherwise. We found that SIMD instruction, expensive instruction, and branch miss prediction throughput scales linearly with the CPU frequency, like the compute-bound microbenchmark. We also found that expensive instructions like division are often 50 times slower than normal instructions — which corresponds roughly with Agner Fog's findings [Fog14] on the same processor architecture (21–296 times slower) — with an instruction retire rate similar to the memory-bound workload (0.04 vs 0.03 instructions retired per cycle) illustrating that *low instruction retire rates are not good indicators for memory-bound workloads.*

3.1.3 Implementation

This section describes the implementation details of `pappeadapt` for the purposes of reproducibility and full disclosure.

`pappeadapt` is implemented in C++. `pappeadapt` uses the Linux userspace power governor, changing the CPU frequency by writing to the sysfs files `/sys/-devices/system/cpu/cpu*/cpufreq/scaling_setspeed`. `pappeadapt` assumes that enabling Turbo Boost will yield the same CPU frequency and power consumption as the compute-bound microbenchmark — an assumption which does not hold in general, since Turbo Boost by design does nothing on workloads with high base power consumption, such as memory-bound workloads. `pappeadapt` measures cache misses and instructions retired with Linux `perf_-event_-open` as follows:

- Cache misses (`PERF_COUNT_HW_CACHE_MISSES`) and instructions retired (`PERF_COUNT_HW_INSTRUCTIONS`) are measured in one group.
- All applications are measured (`pid == -1`).
- We measure each CPU separately.

`pappeadapt` requires super user permissions, for two purposes, (1) setting the CPU frequency, and (2) measuring all applications.

3.2 Evaluation

This section evaluates the energy savings from using `pappeadapt` and statically reduced CPU frequencies. This evaluation aims to answer 4 questions:

1. How well do the static and adaptive CPU settings perform in terms of energy cost (E, ED, and ED^2).
2. When do the settings perform well?
3. Do the microbenchmarks exemplify the most extremely compute and memory-bound workloads?
4. How can we improve `pappeadapt`?

We evaluated the techniques on the machines in Table 3.1 with 54 benchmarks from 4 benchmark suites:

SPEC2006 The premiere sequential benchmark suite using the "reference" inputs. This suite is frequently used to evaluate realistic system performance and compiler optimizations impact.

PARSEC 3.0 A set of parallel programs using the "native" input sets. We excluded the benchmarks `dedup`, `facesim`, `raytrace`, and `vips`, either because the benchmarks did not have the "native" input set, or because they depended on libraries which were unavailable on our machines.

SPLASH-2x A set of highly optimized parallel programs maintained by the PARSEC authors, using "native" input sets. We excluded `fft`, because it requires too much memory, we also excluded `lu_cb`, `lu_ncb`, and `radix`, because their execution times were below 10 seconds. The benchmark suite was originally written in the 1990's, and is frequently used in computer architecture studies.

NPB 3.3.1 A set of parallel benchmarks written by NASA , using the "B" input sets. We excluded `ep`, `is`, and `mg`, because their execution times were below 10 seconds. The benchmark suite was initially written in the 1990's, but it has been updated several times since, and it is frequently used to evaluate the performance of enterprise machines and supercomputers.

We recorded the execution time with `time` and energy consumption with the Linux sysfs interface for Intel RAPL. Each of the measurements was repeated at least 5 times and all presented measurements are either median values, or geometric mean of median values. We use medians, rather than arithmetic means, because medians are more resistant to outliers.

3.2.1 Energy savings

Figure 3.5 and Table 3.2 show the relative cost of the different settings in terms of E, D, ED, and ED^2. The costs are relative to the cost at the 4 GHz (4 GHz with Turbo Boost), the default behavior for most Linux power governors. Each small dot represents the median cost for a benchmark, and the larger points show the geometric mean of the median costs.

Adaptively and statically reducing `xeon`'s CPU frequency gave energy savings above 40 %. Statically reducing the CPU frequency to 1.6 GHz and 1.8 GHz gave the best energy savings of 41.6 %, while increasing the delay by 103 and 83 % respectively. Adaptively optimizing for E gave the second best energy savings at 41.5 %, while increasing the delay by 62.5 %. Statically reducing the CPU frequency to 2.2 GHz gave energy savings of 40.3 %, larger savings than we predicted possible from the microbenchmarks. The larger savings are caused by the parallel benchmarks (see Section 3.2.2).

Adaptively and statically reducing `xeon`'s CPU frequency gave ED savings

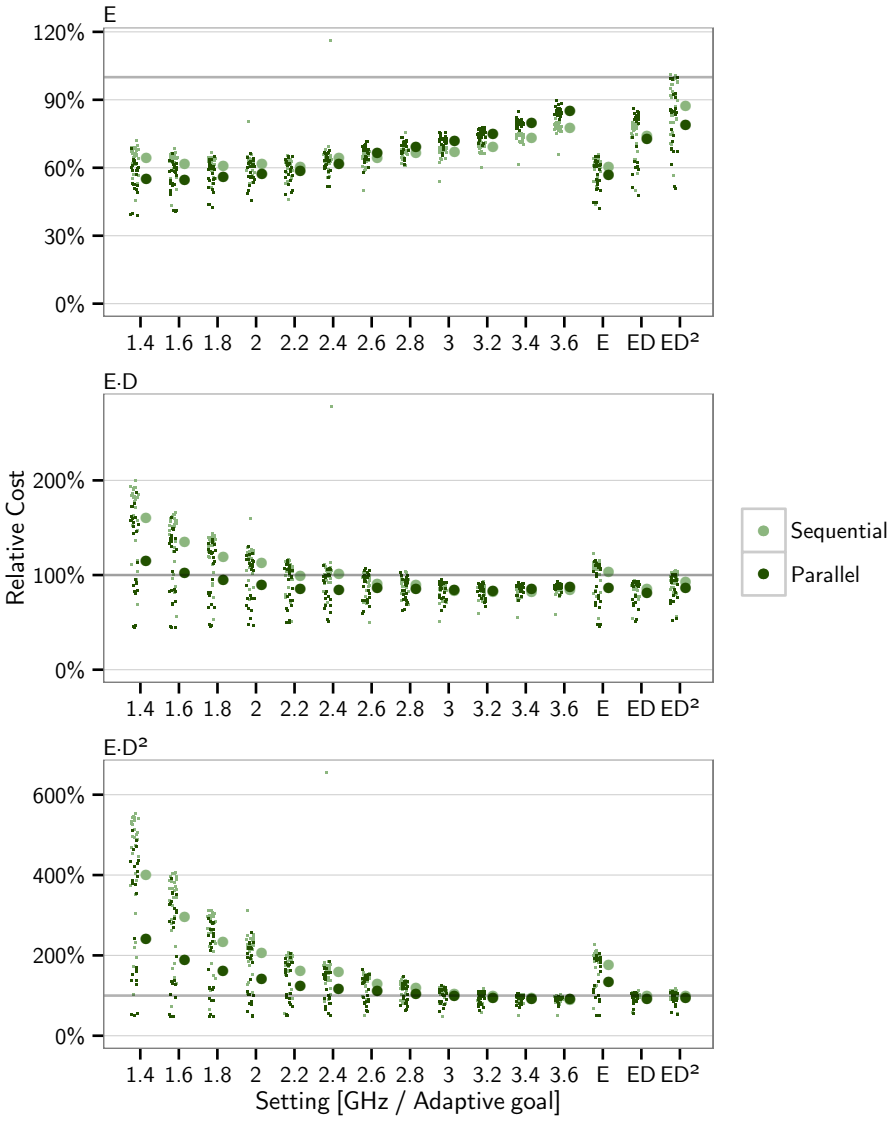


Figure 3.5: Energy costs for static CPU frequencies and adaptive settings. Shows median energy costs (small dots) and geometric mean of median energy costs (points). Lower is better.

Setting	D	P	E	ED	ED^2
Adapt E	162.5 %	36.0 %	58.5 %	95.2 %	154.8 %
Adapt ED	114.0 %	64.4 %	73.4 %	83.6 %	95.3 %
Adapt ED^2	107.6 %	77.4 %	83.3 %	89.7 %	96.5 %
1.4 GHz	230.0 %	25.9 %	59.7 %	137.4 %	316.3 %
1.6 GHz	203.0 %	28.7 %	58.4 %	118.5 %	240.5 %
1.8 GHz	183.4 %	31.8 %	58.4 %	107.1 %	196.5 %
2 GHz	170.0 %	35.1 %	59.7 %	101.4 %	172.3 %
2.2 GHz	155.1 %	38.4 %	59.6 %	92.5 %	143.5 %
2.4 GHz	147.5 %	42.8 %	63.1 %	93.2 %	137.5 %
2.6 GHz	136.0 %	48.1 %	65.4 %	88.9 %	121.0 %
2.8 GHz	129.0 %	52.5 %	67.7 %	87.4 %	112.7 %
3 GHz	121.1 %	57.1 %	69.2 %	83.8 %	101.5 %
3.2 GHz	115.6 %	62.1 %	71.8 %	83.0 %	96.0 %
3.4 GHz	110.3 %	68.9 %	76.1 %	83.9 %	92.6 %
3.6 GHz	105.7 %	76.6 %	81.0 %	85.7 %	90.6 %
4 GHz	100.0 %	100.0 %	100.0 %	100.0 %	100.0 %

Table 3.2: Geometric mean of median energy and delay products normalized for adaptive and static CPU frequencies, relative to the cost at 4 GHz. Lower is better.

above 14 %. Statically reducing the CPU frequency to 3.2 GHz gave the best ED savings of 17 %. Adaptively optimizing for ED gave the second best ED savings at 16.4 %. Statically reducing the CPU frequency to 3.6 GHz (f_{hi}) gave ED savings of 14.3 %, while reducing the CPU frequency to 1.8 GHz (f_{lo}) increased ED by 7.1 %. The poor performance of f_{lo} corresponds to our expectations from the microbenchmarks; statically optimizing the CPU frequency for memory-bound workloads increases the delay prohibitively in compute-bound workloads. In other words, if you want to optimize for ED, and do not know which workload you have, either optimize adaptively or optimize statically based on a mostly compute-bound workload.

Adaptively and statically reducing `xeon`'s CPU frequency gave ED^2 savings below 10 %. Statically reducing the CPU frequency to 3.6 GHz gave the best ED^2 savings of 9.4 %. Statically reducing the CPU frequency to 1.8 GHz (f_{lo}) increased ED^2 by 96.5 %, approximately half of the worst case performance we saw in the memory-bound microbenchmark. Adaptively optimizing for ED^2 saved 3.5 %, while adaptively optimizing for ED saved 4.7 %. The performance gap between the techniques is largest in parallel benchmarks and smallest in sequential benchmarks (SPEC2006). We believe that optimizing for ED performs better than optimizing for ED^2, because of the cost of the experimental steps. The former technique experiments using $f_{hi} = 3.6$ GHz, which is 9.4 % better

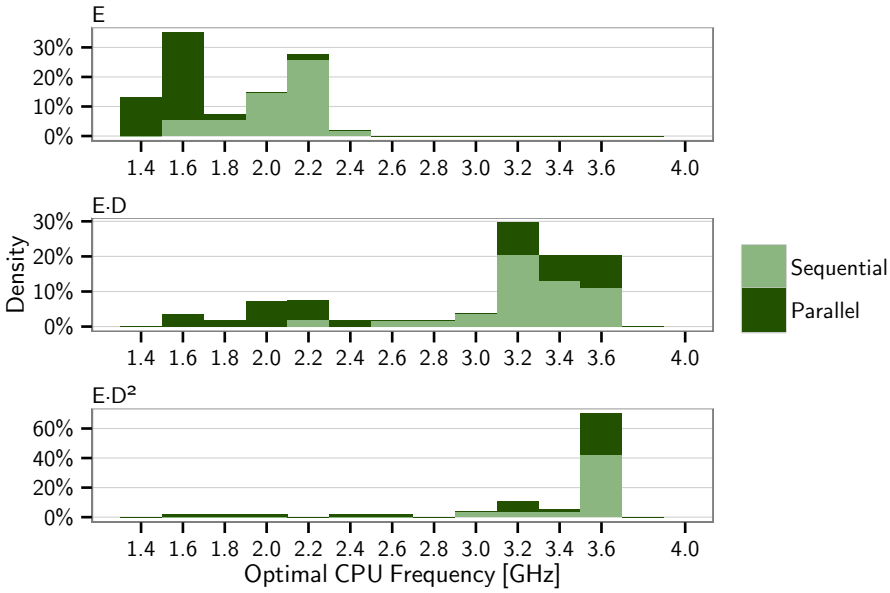


Figure 3.6: Distribution of optimal static CPU frequencies for individual benchmarks. Higher is better.

on average than the latter technique's $f_{hi} = 4\,\text{GHz}$. This indicates that our technique should select experimental CPU frequencies more carefully, to reduce the cost of experimenting.

Reducing the CPU frequency adaptively or statically can dramatically improve ($>$40 percent) energy efficiency, somewhat improve ($>$14 %) energy delay products (ED), and slightly improve ($<$10 %) energy delay squared products (ED^2), as we expected from the microbenchmarks.

3.2.2 Sequential and parallel performance

Figure 3.6 shows how frequently the settings are optimal. Based on the microbenchmarks we would expect all optimal CPU frequencies should be between 1.6 GHz and 2.2 GHz when optimizing for E, between 1.8 GHz and 3.6 GHz when optimizing for ED, and between 1.8 GHz to 4 GHz when optimizing for ED^2. It turns out that parallel workloads can benefit from lower CPU frequencies than we expected.

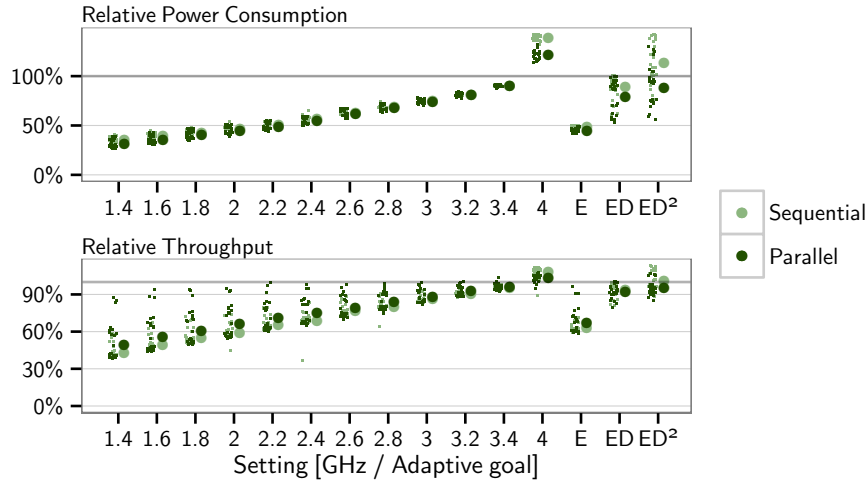


Figure 3.7: Power consumption and execution time relative to 3.6 GHz. Shows median energy costs (small dots) and geometric mean of median energy costs (points). Lower is better.

Our initial expectations hold for the sequential benchmarks: All sequential benchmarks, with one exception, reach optimal energy efficiency in the expected range, with 2.2 GHz being the most frequently optimal. The exception is `401.bzip2`, which saves 36.8 % energy at 2.4 GHz compared to 36.4 % at 2.2 GHz, a small difference which could be a fluke. All of the sequential benchmarks reach an optimal ED and ED^2 efficiency in the ranges 2.2-3.6 GHz and 3.0-3.6 GHz, respectively. The statically optimal CPU frequencies for ED and ED^2 fit in a considerably smaller ranger than we predicted from the microbenchmarks, presumably because realistic benchmarks are not as compute or memory-bound as our microbenchmarks.

The parallel benchmarks generally benefit more from using low CPU frequencies: 1.6 GHz is the most frequently optimal CPU frequency, and 6 of the benchmarks reach optimal energy efficiency at 1.4 GHz. Figure 3.7 illustrates that parallel benchmarks are more sensitive to the CPU frequency, both in terms of power consumption and execution time: increasing the CPU frequency reduces execution time less, and increases power consumption more, than increasing the CPU frequency in sequential workloads. Both of these observations can be explained easily:

- In a parallel workload, the CPUs consume more power relative to shared components, such as caches, increasing the dynamic power consumption.

- In a parallel workload, the shared components, such as caches and memory, are more likely to be limiting resources, decreasing the speedup from higher CPU frequencies.

The most frequently optimal CPU frequencies for ED and ED^2 are 3.2 and 3.6 GHz respectively, for both parallel and the sequential benchmarks. The sequential benchmarks tend to benefit more from CPU frequencies above 3.2 GHz than the parallel benchmarks.

3.2.3 Comparison of benchmarks and microbenchmarks

Our initial assumptions about workloads appear to hold for the sequential benchmarks, indicating that the microbenchmarks represent extreme cases of compute and memory-bound workloads: The optimal static CPU frequencies for E, ED, and ED^2 are in the ranges we predicted based on the benchmarks.

Our initial assumptions about workloads do not hold for the parallel benchmarks, since several of the benchmarks perform better at lower CPU frequencies than we predicted. Increasing the CPU frequency in the parallel benchmarks increases power consumption more, and generally reduces execution time less than increasing the CPU frequency in the sequential benchmarks (see Figure 3.7).

3.3 Related work

In this section we will give a brief review — and relate to — prior works on (1) workload aware power governors, (2) studies of CPU frequencies, and (3) techniques for optimizing for energy efficiency. We only include power governors for fully utilized systems, since they are significantly different from other power governors, and we focus mostly on recent works, because the power-performance tradeoffs have changed significantly over time.

3.3.1 Power Governors

As far as we know, there are currently only 3 power governors with published papers — Koala, Green Governors, and eDVFS — all of which are implemented for Linux.

Koala by Snowdon et al. adaptively optimizes energy efficiency according to a generalized ED^N energy cost [SLSPH09]. Koala predicts power consumption and delay with a linear regression model based on performance counters. The performance counters are selected by training on SPEC2000 benchmarks, which took them approximately 18 days. The resulting models are individual for each processor, but they generally used cache activity and miss rates, together with the number of stalled CPU cycles, similar to Green Governors.

Koala generally yielded significant ED and E savings on SPEC2006 relative to using the highest CPU frequency, but on some benchmarks it performed significantly worse than the highest CPU frequency. The most extreme case is the LBM benchmark, where they over approximated execution speed by 25 %, and under approximated energy use by 20 %. As a result Koala used a higher CPU frequency than it should have.

Green Governors by Keramidas et al. [KSK10] adaptively predicts optimal CPU frequencies based one of two models: (1) a stall based model which requires a count of the number of stalled CPU cycles due to cache misses and (2) a miss based model, which requires a count of non-overlapping cache misses. Both of the required performance counters do not exist for common processors, but they later proposed [SKK11] another model which only used commonly available performance counters: Estimate the stall cycles due to cache misses as the minimum of (1) the number of CPU stall cycles and (2) the average cache miss latency multiplied by the number of cache misses $\min(\mathit{frontendStalls}, \mathit{cacheMisses} \cdot \mathit{avg}_{\mathit{cacheMissLatency}})$.

They used the updated performance model together with power consumption models of Goel et al. [GMG+10] to implement *Green Governors*, which optimizing ED or ED^2. The Green Governors generally yielded significant ED and ED^2 savings on SPEC2006 relative to using the highest CPU frequency, but on some benchmarks it performed significantly worse than the highest CPU frequency.

eDVFS by Kim et al. [KEYM14] adaptively optimizes energy efficiency. They model execution time as the maximum of the memory's execution time — estimated from bandwidth requirements — and the processor's execution time — estimated from an equation system which involves memory latency, cache misses, and CPU frontend stall cycles. eDVFS approximates a solution to the equation system by evaluating it iteratively 10 times, or until the values converge. The equations in the system state three relations (1) the number of cache misses increases with fewer CPU frontend stalls, because then the processor can

make more memory accesses, (2) the number of CPU frontend stalls increases with lower memory latencies, and (3) the memory latencies increase with higher cache miss rates.

They model power consumption as the sum of the memory's power consumption, as a linear function of the cache miss rate, and the processor's power consumption, as a function of CPU utilization and frequency. The power consumption and the execution time models need to know how memory latency and bandwidth relate processor frequency and cache misses, which eDVFS detects through a training step. eDVFS yielded significant E savings on SPEC2006 compared to the highest CPU frequency, but the savings were often significantly worse than the power savings achieved by statically optimizing the CPU frequency.

`pappeadapt` fundamentally differs from the prior workload aware power governors in two ways:

1. Tool uses online experimentation, instead of making predictions directly from performance counters
2. Tool prunes the set of possible CPU frequencies based on static training, to minimize poor choices of CPU frequencies.

The differences were designed to improve the robustness of `pappeadapt` — to reduce the number of cases where `pappeadapt` performs poorly — a goal which we largely achieved, since `pappeadapt` always yields energy delay product savings near the static optimal frequencies. Robustness is important because it increases our confidence in the power governor; if it performed horribly in a few of the benchmarks we evaluate on, then how can we be sure it will behave nicely in general?

The less robust power governors — Koala and Green Governors — essentially predict execution time by assigning a weight to each performance counter. In our initial studies (Section 3.1.2.1), we pointed out that the individual performance counters are poor indicators of how execution time will scale with the CPU frequency. An application `A` can literally cause 10 times as many cache misses as an application `B`, while `B` is more memory-bound than `A`. eDVFS's evaluation showed more robust results, possibly its execution time model is very non-linear, or possibly because only used the training step to tune fundamental metrics — memory bandwidth and latency — rather than performance counter weights.

In our quest for a robust power governor, we have also made some conservative decisions which limit our potential savings, in three ways: (1) our model of

power consumption is very simple and conservative — the power models used by eDVFS could probably improve our predictions, (2) pruning the set of possible CPU frequencies based on sequential training applications is disregards low CPU frequencies which are optimal for some parallel workloads.

Our online experimentation is heavily inspired by Mars et al.'s Shutter technique [MVHS10]. The Shutter technique adaptively sets the number of threads used parallel batch applications based on periodic experiments in 3 repeating steps:

1. Use 1 thread for 0.1 s, while measuring throughput.
2. Use as many threads as possible for 0.1 s, while measuring throughput.
3. Either use 1 thread, or as many threads as possible for 0.8 s, based on which solution performed best.

Our technique, illustrated in Listing 3.1, follows the same basic outline, but it has 2 domain specific improvements: (1) We predict the optimal setting based on a variation of Amdahl's law, and (2) we avoid experimenting when we are confident that the highest setting is optimal.

3.3.2 Studies in CPU frequencies impact on power and performance

Le Sueur and Heiser investigated the potential energy savings from reducing CPU frequency on 3 computers with AMD processors produced in 2003, 2006, and 2009 [LSH10]. They found that the computers from 2003 and 2006 could save approximately 10 and 30 percent energy respectively by reducing their CPU frequencies by 400 MHz, on the SPEC2000 benchmark 181.mcf, while the 2009 computer was most energy efficient at its highest CPU frequency. They argued that the 2009 computer did not benefit from using lower CPU frequencies for three reasons: (1) the lower feature size in the 2009 computer means that it has a higher static power consumption, (2) the new computer has better memory performance, due to faster RAM, larger caches, and hardware prefetching, which effectively makes the benchmark less memory-bound, and (3) the new computer has 4 cores, which complicates voltage scaling. Another possible reason for the low benefits on the newer machine, is that the processor consumed a smaller fraction of the entire computers power consumption, than on the older computers.

Le Sueur and Heiser later performed a similar survey [LSH11] with three processors from 2009: an OMAP4430 (ARM A9), an Intel Atom Z550, and an Intel i7-870. The latter survey investigated two workloads with low CPU utilization — a video player and web server workload — as well as a workload with high CPU utilization —SPEC JBB2005 with 4 warehouses. Reducing the CPU frequency in the workloads with low CPU utilization significantly reduced the Intel Atom and OMAP4430 based computers' energy consumption, but marginally increased the Intel i7 based computer's energy consumption. Meanwhile, reducing the CPU frequency in SPEC JBB2005 significantly improved its energy efficiency.

Schöne et al. [SHM12] studied 7 different high end computers' memory and L3 cache bandwidth at various CPU frequencies on a parallel microbenchmark. They found that the relationship between both memory bandwidth and CPU frequency is machine specific, as well as the relationship between L3 bandwidth and CPU frequency. On most of the computers the cache bandwidth grew linearly with the CPU frequency in the single threaded case, but sublinearly in parallel case because the cache bandwidth was nearly exhausted, even at low CPU frequencies. Schöne et al.'s findings highlight that different machines become memory-bound at different cache miss rates and cache access rates; a further motivation for using training steps in power governors.

Etinski et al. [ECLV12] performed a study of how the CPU frequency influences execution time of the MPI version of the NAS benchmark suite on a cluster of 30 computers with 24 cores each. They found that universally — for all the benchmarks — the had execution times depended less on the CPU frequency when using more parallelism, because CPU frequency scaling does not affect communication time, which is significant in parallel workloads. Their findings illustrate that reducing the CPU frequency is a more viable approach in highly parallel workloads.

Kambadur and Kim [KK14] studied the impact of various compiler optimizations, manual optimizations, parallelism, processor idle states, CPU frequencies, and power governors. On the optimization front they found that parallelizing code could save 50 % energy, while other manual optimizing only improved energy efficiency in one benchmark out of eight. On the power governor front, they found that (1) reducing operating frequencies increased energy consumption, while disabling TurboBoost reduced energy consumption in some cases — especially on parallel code — (2) the Linux power governors ondemand and performance are approximately equally energy efficient when the CPU is fully utilized, and (3) processor idle states saves $10 - 19$ % energy — the largest savings occur when TurboBoost is enabled.

Our results have shown a larger benefit from using lower CPU frequencies than

the findings of Kambadur and Kim, as well as the findings of Le Seur and Heiser, a difference which we believe is caused by two factors:

1. Our evaluation only measures processor power consumption, not the power consumption of the entire computer.
2. We evaluated on different machines.

Overall, both our findings and those of prior work two things: (1) parallel workloads benefit more reducing CPU frequencies than sequential workloads and (2) the benefits from reducing CPU frequencies are very machine dependent. The machine specific features energy features is the fundamental reason why `pappeadapt` and the other workload specific power governors have a training step.

3.3.3 Optimizing software for energy efficiency

Power-Sleuth is a power consumption profiler by Spiliopoulos et al. [SSK12] capable of predicting applications power-performance characteristics at different CPU frequencies based on a single profile run. The predictions are made for individual phases of the program's execution, by first identifying phases with the application ScarPhase [SEH11]. Power-Sleuth predicted the performance of each phase with the execution time model of Green Governors, and the power consumption with a trained linear regression model based on performance counters similar to Koala's power model.

3.4 Future work

`pappeadapt` is a promising power governor which has not yet reached its potential, in its current form. Adaptively setting CPU frequencies with `pappeadapt` does not outperform the best static assignments of CPU frequencies, but we believe that goal is reachable with a few improvements. Reaching that goal would make `pappeadapt` an attractive means for reducing energy consumption in a wealth of computer systems, from data centers to personal computers and smart phones. However, the model for predicting memory-bound workloads execution times is very precise, in its current form. We believe that the model could be used for workload characterization, to aid in selecting the right system configurations for different workloads, in data centers and embedded devices.

We have identified 5 means for improving `pappeadapt`'s performance:

1. Train for parallelism.
2. Improve training robustness.
3. Reduce experimentation cost.
4. Improve measurement precision.
5. Improve prediction accuracy.

Out of these potential improvements, we believe that improving training robustness, reducing experimentation cost, and improving measurement precision will give the largest benefit, and require the least effort.

Train for parallelism. We could improve `pappeadapt`'s performance in parallel workloads by detecting parallel workloads and training with parallel microbenchmarks. `pappeadapt` uses estimates of relative power consumption from the sequential microbenchmarks, which are not representative of power consumption in parallel workloads. `pappeadapt` actually performed better in parallel workloads, despite being trained in sequential workloads, indicating that our current approach is more profitable in parallel workloads. We believe the best way of detecting parallel workloads is either:

1. Tracking the number of threads spawned for computation by instrumenting the systems threading libraries.
2. Tracking the number of active threads (`loadavg`) by querying the operating system.

We believe that instrumenting the systems threading library is the better option here, since operating system statistics are normally accumulated over several minutes, making them poor indicators of current parallelism.

Training output reliability. The accuracy of the training output is vital for our `pappeadapt`'s performance. Inaccurate training output skews the estimated power consumption — causing `pappeadapt` to optimize based on wrong estimates — and changes the CPU frequencies that `pappeadapt` will experiment with and predict performance for. We can improve the training outputs reliability by running the traning phase repeatedly, until the mean of all output

variables coverge: This will increase the training phase's length from approximately 2 minutes, to perhaps 20 minutes, which we consider to be worth the effort.

We know that the training output is not sufficiently accurate, because we received significantly different training output when repeating the training phase after the experiments: In our preliminary experiments on the microbenchmarks, we found that the lowest optimal CPU frequency was 1.8 GHz when optimizing `xeon` for ED or ED^2, but the repeated experiments found that 2.0 GHz and 2.2 GHz were optimal. The error in the training output meant that `pappeadapt` experimented with CPU frequencies which were lower than necessary, when optimizing for ED and ED^2.

Reducing experimentation cost. The adaptive procedure experiments with the CPU frequencies f_{hi} and f_{lo} 20 % of the time. Experimenting with f_{lo} is particularly expensive when optimizing for ED^2 on compute-bound workloads, because it increases execution time significantly. We can reduce the cost by (1) experimenting with CPU frequencies near the predicted optimal frequency, and (2) deriving a smallest possible optimal CPU frequency based on the cache miss rate.

Improving measurement precision. We predict which CPU frequency is optimal based on measurements of instruction retire rates of all applications on all CPUs. Unfortunately, the measurements are rather imprecise: Measuring the instruction retire rate with a static CPU frequency at 0.1 second intervals, should show phase behavior [SEH11] — extended periods of time where the measurements do not change significantly, because the workload does significantly when it is in a phase. We found that successive measurements typically varied with 20 %, never showing phase behavior. If the instruction retire rate typically varies with 20 % when we are not changing the CPU frequency, then the impact of changing the CPU frequency will be less visible — reducing the signal to noise ratio of the measurement we use for our predictions.

We can improve measurement precision by (1) increasing measurement durations, sacrificing the ability to quickly adapt to changing workloads, or (2) finding more accurate ways to measure throughput. We previously spawned the benchmarks from `pappeadapt`, and measured their instruction retire rates through inherited perf_event handles, which seemed more accurate, but would be an impractical to use in real systems. We believe that a significant portion of the measurement errors are caused measuring with grouped perf_event counters for all processes on each individual CPU. Measuring on each individual

CPU may be deferred to context switches. If that is the case, we should either measure after context switches, or compensate for the unmeasured execution time.

Improving prediction accuracy. Our prediction model is accurate (see Figure 3.4) but simplistic. We could improve prediction accuracy by modelling overlapping of on-chip and off-chip activities, modelling that that the CPUs can hide a number of cycles of latency. Improving prediction accuracy seems like overkill for the current setup, where prediction accuracy is significantly better than the measurement precision, but it may be necessary if we experiment with a smaller range of CPU frequencies, extrapolating from the model.

We believe these potential improvements could the adaptive techniques more attractive, hopefully outperforming the static optimal CPU frequencies. Even though we rarely outperform the static optimal CPU frequencies in individual benchmarks, we believe our adaptive technique is an attractive alternative to prior work on optimizing CPU frequencies for memory-bound workloads, and we believe our model of execution time in memory-bound applications could be applied in other situations.

3.5 Concluding remarks

Tuning CPU frequencies can save a lot of energy. We have experimentally shown that we can save 41.6 %, and 40.3 % energy by statically restricting the CPU frequency to the CPU frequencies to the CPU frequencies which are optimal for compute and memory-bound microbenchmarks, respectively. Statically using those CPU frequencies is an easy way to reduce energy consumption (See Section 3.2.1). It is harder to optimize ED and ED^2 by statically reducing the CPU frequency, because they exhibit a larger difference in which CPU frequencies are optimal. With `pappeadapt` — a power governor which adaptively experiments with CPU frequencies, measures how the computer performs, and adjusting the CPU frequency accordingly — we reduced the processors geometric mean energy consumption by 41.5 %, ED by 16.4 %, and ED^2 by 4.7 %, on the benchmark suites NPB 3.3.1, SPEC2006, PARSEC 3.0, SPLASH-2x. By adaptively experimenting and optimizing CPU frequencies, `pappeadapt` achieves consistent energy product savings for a wide range of workloads.

`pappeadapt` differs from prior power governors by making predictions from experiments, rather than models driven by cache miss rates and CPU frontend stall cycles. Predicting directly from performance counters is attractive because

it is essentially free — unlike our experiments with the CPU frequency — but the predictions cannot be accurate for all workloads, because the performance counters can misrepresent how memory-bound workloads are by an order of magnitude (See Section 3.1.2.1). Performing the experiments has a cost, but it allows `pappeadapt` to work consistently.

`pappeadapt` predicts execution times by modelling and estimating the proportions of on-chip activities, which speed up linearly with the CPU frequency, and off-chip activities, which are independent. The model allows us to accurately predict the execution time at any CPU frequency from execution time measurements at two different CPU frequencies. Such predictions are not just valuable for power governors, but could also be used to select a computers processor or CPU frequency, based on the throughput requirements and the workload it is have.

Overall, `pappeadapt` illustrates how we can adaptively improve energy efficiency for the current workload by reducing CPU frequencies — increasing execution time and decreasing power consumption. In the next chapters we will investigate the opposite approach, improving energy efficiency by reducing execution time, even if that means increasing power consumption.

Chapter 4

Minimizing Blocking with Transactional Memory

In the previous chapter we saved energy by reducing the processors voltage, with a middleware solution exploiting DVFS and performance counter present in modern computers. In this chapter we extend the energy saving efforts by reducing execution times of parallel application, with a middleware solution exploiting transactional memory support present in modern computers.

Parallel applications use synchronization to coordinate asynchronous threads. Synchronization is not free. Depending on the structure of the parallel application, how much parallelism it exposes, and how the work is distributed, threads may spend much time blocked while synchronizing — time that *should* be spent on useful work.

Transactional Memory (TM) can be used to reduce blocking: Rather than waiting for a lock to be acquired, the thread enters a transaction and attempts to execute the protected code without a lock. Should a data-race occur between transactions, some of them will be aborted — reverting all of the transactions writes to memory, hiding the data-race — such that the transaction can safely be restarted. Individual transactions have a higher performance overhead than regular locking schemes, but transactions allow for parallel executions of critical sections, potentially reducing execution time [BCKT14, SATH+06] and energy

consumption [MBH05].

Hardware Transactional Memory (HTM) has recently been adopted by hardware manufacturers in architectures such as Intel Haswell and IBM Power8 [CMF+13, Int14b]. Hardware supported transactions have relatively low overhead, but they can fail both spuriously and deterministically, even without data-races. Using HTM requires a backup plan for all transactions, in case they fail repeatedly.

This chapter introduces:

- HTM based methods for executing `barriers`, `critical` sections and `taskwait` without blocking or extra programmer intervention:
  - The speculative method for critical sections is more resistant to contention than prior work.
  - Unlike existing speculative methods for barriers, our methods do not require non-transactional memory accesses within transactions.
- An evaluation of aforementioned methods on a series of microbenchmarks and the Barcelona OpenMP Task-Suite, explaining why and when they work.

This chapter is structured as follows: Section 4.1 describes how to speculate critical sections, taskwaits and barriers, and Section 4.2 evaluates the performance of the methods. Section 4.3 describes limitations of the methods, and how they relate to prior work and extension proposals for OpenMP. Finally Section 4.4 summarizes our findings.

4.1 Avoiding Blocking in OpenMP

This section describes how we use HTM to minimize blocking time in locks, barriers, and taskwaits. We access HTM capabilities through three functions:

`tbegin(LABEL)` Start a transaction. If the transaction fails, it will roll back all its changes and go to the label `LABEL`.

`tend()` Commit the transaction, atomically revealing its changes to all other threads.

`tabort()` Force the transaction to fail.

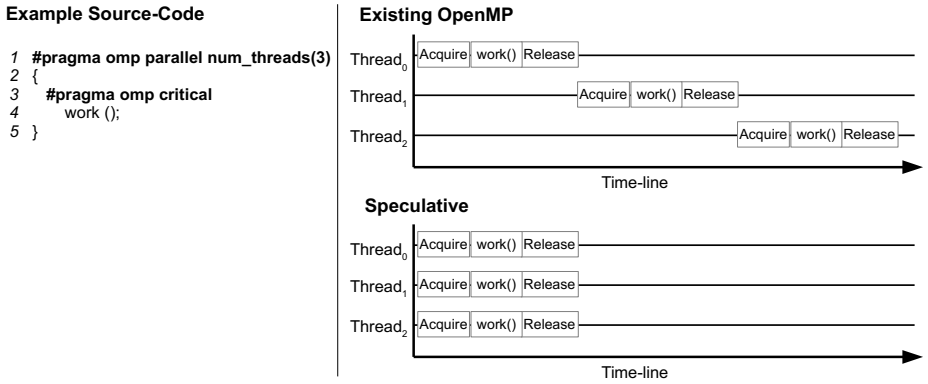


Figure 4.1: Code and timelines for computation with 3 threads using a critical section, illustrating the difference between the existing OpenMP approach and our speculative version.

The functions correspond to the subset of TM capabilities in IBM Power8 and Intel Haswell processor architectures.

4.1.1 Critical sections

Critical sections are typically used to update shared variables in parallel code, for instance updating counters or data structures. Critical sections acts as a serialization point, where only one thread can execute the protected region at any one time. Figure 4.1 shows an example where three threads encounter a critical section. The threads acquire the critical section's lock before entering the critical section. Thread_0 acquires the lock first, which traditionally means that Thread_1 and Thread_2 will have to block until the lock is released. With *lock elision*, threads can speculatively ignore the lock acquisition, and avoid blocking.

Listing 4.1 illustrates our lock implementation, which supports *lock elision*, and truncated exponential backoff [And90]. To avoid blocking, lock elision attempts to use transactions rather than regular locks: Instead of acquiring a lock, we start a transaction (Line 4), and commit the transaction instead of releasing the lock (Line 21). If the transaction fails three times in a row, we fall back to using a test-and-set lock (Lines 8 and 23), and do not let any transactions commit if the lock is held (Line 20). Lock elision follows normal lock semantics: successful transactions appear to execute atomically while the lock is released and failed transactions have no visible side effects.

Multiple transactions can execute a single critical section in parallel as long as the transactions succeed. Unfortunately, transactions do not always succeed. Transactions are less likely to succeed if the lock is held frequently. When transactions fail repeatedly, they fall back to using the underlying traditional lock, creating a harmful feedback loop known as the lemming effect [DLM+09]: once a few lemmings jump off a cliff (transactions fail), the other lemmings will follow suit. If a few consecutive transactions fall back to the underlying lock, then all concurrent transactions will most likely fail, and possibly fall back to using the underlying lock.

We mitigate the lemming effect by (1) not counting aborted transactions (Line 5) and (2) using a truncated exponential backoff variant, which is highlighted green in Listing 4.1. The lemming effect occurs when the lock is contended. Truncated exponential backoff adaptively avoids the lemming effect, by backing off when the lock is contended, and the transaction is unlikely to succeed.

We use a truncated exponential backoff with a slot size $a = 1024$ processor cycles, and a truncation of $b = 2^{8+\lfloor \log_2 t \rfloor}$, where t is the number of threads. Each thread has a `delay` variable (Line 14), which indicates how contended the lock is. Before acquiring the lock, threads must backoff (wait) for a number of clock cycles sampled randomly from 0 to $2^{\texttt{delay}}$, unless `delay` is 0 (Line 2). Our lock implementation is largely based on another form of lock elision called Speculative Lock Removal (SLR) [ALM14]. Our technique differs from SLR in how we mitigate the lemming effect.

```
1  void acquireLock(lockVar) {
2    backoff();
3    for(int a=0; a<3; a++) {
4  FOR:tbegin(ERR); return;
5  ERR:if(tAborted()) goto FOR;
6      if(tCannot()) break;
7    }
8    if(tryLock(&lockVar))
9      return;
10   delay = min(delay + 1, 0);
11   acquireLock();
12 }
13
14 __thread unsigned delay;
15 // Wait U(0;a<<delay) cycles
16 void backoff();
17
18 void releaseLock(lockVar) {
19   if(isInTransaction()) {
20     if(isLocked()) tabort();
21     tend();
22   } else {
23     unlock(lockVar);
24   }
25   delay = max(delay - 1, b);
26 }
```

Listing 4.1: Lock with lock elision (bright orange) and exponential backoff (green).

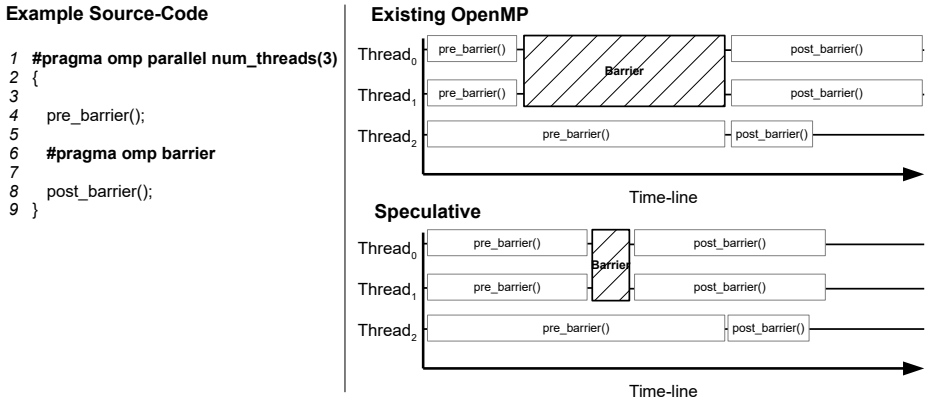


Figure 4.2: Code and timeline illustrating the benefits of speculative execution over existing approaches for barrier synchronization.

4.1.2 Barrier/Taskwait

Barriers are typically used to orchestrate parallel computations with multiple stages, computations such as linear algebra solvers or image processing with multiple stages or filters. A thread that reaches a barrier must block until all threads arrive at the barrier. Figure 4.2 illustrates a computation with three threads ($\text{Thread}_{0\ldots2}$) using a barrier. Thread_2 is the last thread to reach the barrier, which traditionally means that the other threads (Thread_0 and Thread_1) have to wait. With barrier elision, threads can optimistically speculate beyond the barrier to avoid blocking.

Listing 4.2 illustrates the barrier elision: Instead of blocking for the arrival of the remaining threads (Line 3), we start a transaction (Line 6). At the next synchronization point, we check if the other threads arrived after the barrier (Line

```
1  void barrier_wait(count) {
2    fetch_and_add(&count, 1);
3    while(count!=num_threads) {
4      while(task_schedule()) {}
5      spec_val = num_threads;
6      tbegin(RETRY);
7      spec_adr = &count;
8      return;
9  RETRY: {}
10   }
11 }
```

```
12 __thread unsigned* spec_adr;
13 __thread unsigned spec_val;
14
15 // Called by synchronization
16 void handle_spec() {
17   if(spec_adr == 0)
18     return;
19   if(*spec_adr != spec_val)
20     tabort();
21   tend();
22 }
```

Listing 4.2: Single use barrier with speculative support (highlighted lines).

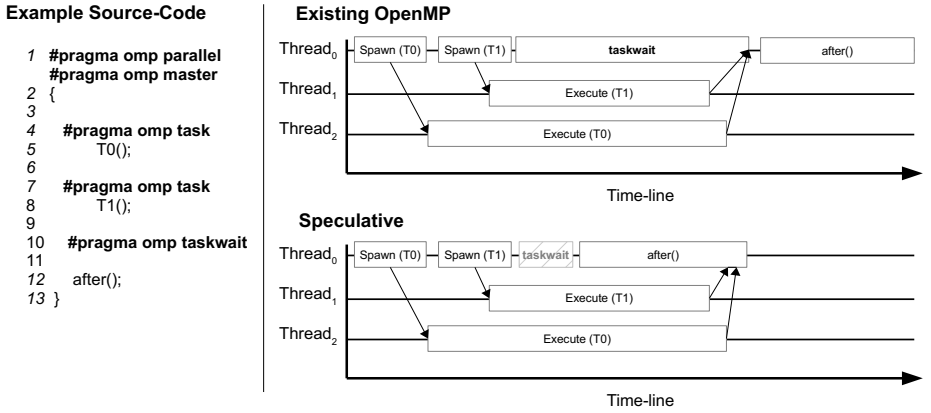


Figure 4.3: Code and timelines illustrating the benefits of speculative execution existing approaches for taskwait synchronization.

19); committing the transaction if all threads arrived (Line 21), and aborting otherwise (Line 20).

The transaction will commit only if it was data-race free and all threads reached the barrier. This ensures that the transaction will appear to execute atomically after all threads reach the barrier.

The barrier in Listing 4.2 is simplified to emphasize barrier elision. The listing's barrier is usable only once and does not synchronize with tasks, whereas our implementation uses sense reversing barriers preceded by taskwait synchronization.

Taskwait synchronization cause threads to block until all children tasks finish, as illustrated by Figure 4.3. We elide taskwait synchronization in the same way as barriers: Attempt to speculate beyond a taskwait if the thread cannot fetch a ready task, and either commit or abort the transaction at the next synchronization.

Our mechanism can speculate across one barrier or taskwait. Ideally, the critical path before the synchronization can run in parallel with critical path after the synchronization, which would halve the makespan.

Table 4.1: Experimental machine

Processor Name	Intel Xeon E3-1276 v3
Processor Configuration	4 cores, 2 hyper-threads/core, 3.6 GHz
Processor Caches	4x32 kB L1D, 4x256 kB L2, 8 MB L3
Compiler	GCC 5.1.0
Operating System	Ubuntu Server 14.04.1 LTS
Kernel	3.17.0-031700-generic

Table 4.2: Task-parallel benchmarks and inputs. We used default cutoffs.

Benchmark	Input-Set
Alignment	prot.100.aa
Fibonacci	48
Floorplan	input.20
Health	large.input
nQueens	13x13 chessboard
Sort	134,217,728 elements
Strassen	4096x4096 matrix size
SparseLU	50x50 matrix, 100x100 submatrix
UTS	tiny.input

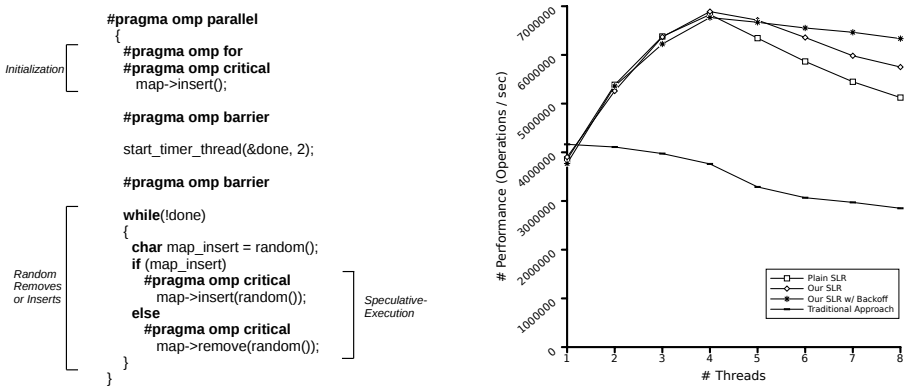
4.2 Evaluation

This section describes the evaluation methodology we used and the results achieved using our proposed implementation.

4.2.1 Experimental Setup

We implemented our speculative methods, described in Section 4.1, in the OpenMP runtime TurboBŁYSK [PBV14]. We evaluated our implementation on the Intel Haswell-based system outlined in in Table 4.1.

Speculative execution across `taskwaits` in task-parallel benchmarks was evaluated using the Barcelona OpenMP Task Suite (BOTS) [DTF+09] with tied tasks and the input sets given in Table 4.2. All benchmark were executed 30 times, and all our results show the median execution time. Cache-statistics were obtained using Linux `perf`, while only counting L3 cache misses outside aborted transactions. BOTS is a benchmark suite where all benchmarks use OpenMP several tasks, and a few `taskwait` directives. This suite was initially written in



(a) Critical section microbenchmark **(b)** Critical section throughput

Figure 4.4: Source code and performance for the critical section microbenchmark.

the late 2000's, and it is frequently used to evaluate the performance of OpenMP libraries on systems with tens or hundreds of cores.

Speculation across OpenMP `critical` directives was evaluated with a microbenchmark that randomly inserts or removes elements from a shared map. We compared the traditional OpenMP `critical` implementations that use locks, a generic software lock elision technique and our improved software-elision technique with and without backoff. The performance metric is the number of map modifications successfully completed per second.

Speculation across OpenMP `barrier` directives was done using a microbenchmark that alternates between barrier synchronization and computation. The microbenchmark is a stress test for barrier synchronization, to illustrate the potential performance improvements from using speculation. The performance metric is the improvement (decrease) in execution time.

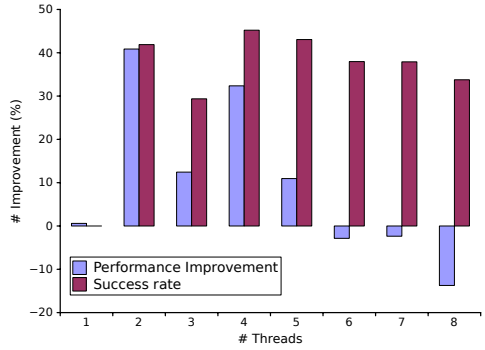
4.2.2 Results

4.2.2.1 Critical section performance

Evaluation of our proposed critical section implementation (see Section 4.1.1) was performed using a common map microbenchmark, which has been used

```
#pragma omp parallel
{
  int tid = omp_get_thread_num();
  for (int i=0; i<1000000; i++) {
    volatile int j;
    for (j=0; j!=10000*((tid + i) % 2); j++) { }
    #pragma omp barrier
  }
}
```

(a) Barrier microbenchmark



(b) Barrier elision improvements

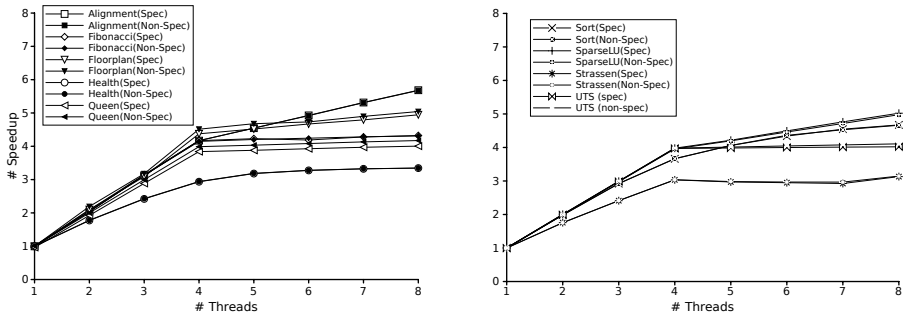
Figure 4.5: Source code and performance for the barrier microbenchmark.

in related work [BER14, DVY14, NM14], illustrated in Figure 4.4:a. A team of threads concurrently operate on a left-leaning red-black tree (GCC's STL map), which initially contains 2^{17} key-value pairs. Half of the operations are insert operations and the other half are remove operations, both using uniformly random keys from $[0; 2^{18} - 1]$. After the threads have operated on the tree for 2 seconds, we record how many operations completed in that time. The tree is protected by a critical section.

Figure 4.4:b shows the sum of the threads throughput for different lock implementations. We evaluate the lock implementation used in GOMP, plain SLR [ALM14], our SLR variant without backoff, and finally our SLR with backoff.

Traditional locking will suffer from a serialization bottleneck because all work in the benchmark is contained within the critical sections. Using lock elision enables the benchmark to scale, but it suffers from the previously mentioned lemming effect: When a few transactions fail, more transaction will quickly follow. Our SLR variant reduces the lemming effect somewhat by not counting failed transactions (Listing 4.1 line 5) and by combining it with exponential backoff the performance degrades more gracefully than the other approaches.

Other variations of this microbenchmark, with different map implementations or distributions of operations, can scale significantly better than this evaluation. We chose this evaluation because it illustrates that there are both benefits (scaling) and disadvantages (the lemming effect) to lock elision.



(a) (b)

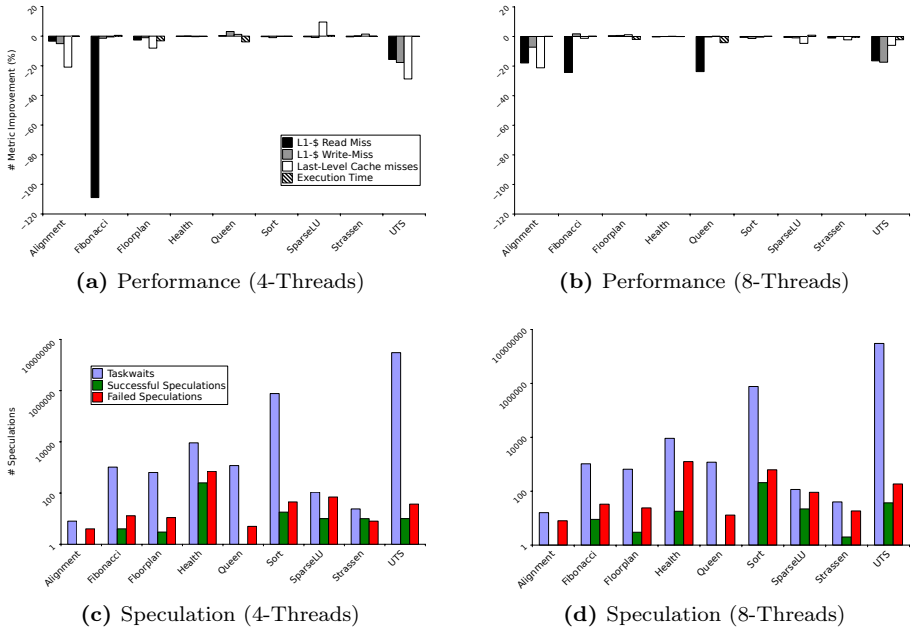
Figure 4.6: Speed-up on the BOTS benchmarks with and without speculation.

4.2.2.2 Barrier/taskwait elision

We evaluate elision of barrier and taskwait synchronization on BOTS, as well as a microbenchmark which illustrates the best case scenario for barrier elision. The microbenchmark is illustrated in Figure 4.5. A team of threads repeatedly enter a barrier. Only half of the threads have any work to do in between the barriers, each thread alternates between having work to do and not.

Figure 4.5 illustrates the performance improvement from speculating across the barriers, and the success rate of the speculations, i.e. the ratio of successful transactions to attempted transactions. The highest performance improvement is 41 % at 2 threads and the highest success rate is 45 % at 4 threads. The performance improvements reduce significantly when using more than 4 threads, presumably because the additional threads are SMT based hyperthreads. Threads which speculative the barrier presumably use more execution resources than threads which wait at the barrier, leaving less resources to their siblings, effectively slowing down the threads which have work to do. The lowest success rate occurs at 3 threads, possibly due to scheduling constraints and TurboBoost: One of the threads is likely running at higher CPU frequency than the other threads, which reduces its chance of specularing successfully.

Figure 4.6 shows the speed-up performance with and without speculation enabled for the BOTS benchmarks. Both the speculative and non-speculative versions follow the trend of linearly scaling up to the number of cores in the system. The performance is degraded when hyper-threading is in use due to the contention for each core's resources. There are no significant differences between the two versions in terms of absolute performance. SparseLU is the only benchmark which showed consistent — although marginal — performance



(a) Performance (4-Threads)

(b) Performance (8-Threads)

(c) Speculation (4-Threads)

(d) Speculation (8-Threads)

Figure 4.7: Speculation and performance deviation statistics on the BOTS benchmark suite with four-threads (a,c) and eight-threads (b,d).

improvements from speculative execution.

Figure 4.7:a-b show memory characteristics and performance of the speculative cases normalized against the non-speculative version. Enabling speculation across taskwaits thrashes the l1 cache significantly — all benchmarks under test experiences this to some extent. Marginal improvements in the last-level cache performance can be seen for the four-thread scenario, where the SparseLU benchmark experiences up to 7 % decrease in last-level cache misses. Overall, compute-bound benchmark that uses heavy divide-and-conquer strategies (e.g. nQueen and Fibonacci) seem to most receptive to negative memory effects when speculating — fortunately, these are benchmarks that cause few last-level cache misses.

Figure 4.7:c-d shows the total number of taskwaits encountered and the total number of taskwait speculations that successfully committed or failed. The divide-and-conquer compute-bound benchmark rarely offers any chance for speculation. For example, even though the nQueen benchmark offers around 1200 barriers and taskwaits, most of them will fail — only 1 speculation managed to successfully retire. Strassen is the most generous application and manages

```
1 void (*functionPtr)();
2 #pragma omp parallel
3 {
4 #pragma omp master
5   functionPtr = puts;
6 #pragma omp barrier
7   functionPtr();
8 }
```

Listing 4.3: A potentially uninitialized function pointer.

successfully speculate more than it fails but with no benefits on execution time. SparseLU is also well behaved, allowing between 10 % (4 threads) to 20 % (8 threads) of speculations to succeed, yielding marginal performance improvements up to 1 %.

The impact on execution time is well below 1 % in most of the BOTS benchmarks. BOTS is a benchmark suite for evaluating of OpenMP run-time systems on systems with tens or hundreds of cores, whereas we our evaluation is limited to a 4 core system. As a consequence, all of the benchmarks expose plenty of parallelism, and the threads hardly ever block. Future system will likely benefit more from speculating over taskwaits and barriers, as threads will be less likely to find work.

4.3 Limitations and Related Work

Our synchronization elision techniques operate on the following principle: Continue executing speculatively when you would traditionally block for events, but check that the event occurred before committing. In other words, we defer blocking until the transactional commit.

This principle is also used for prior work on lock and barrier-elision [ALM14, SON00, MT02], but it is not entirely correct: The principle assumes that the application code does not commit transactions started by the OpenMP runtime, or change the OpenMP runtimes data. This may seem like a small and reasonable limitation, but seemingly innocent application code, such as Listing 4.3, can violate the requirement.

Threads which elide synchronization risk calling uninitialized function pointers (Line 7). An uninitialized function pointer can point to anything, including the OpenMP library code, data or unmapped memory. If the function pointer points to a transactional commit instruction, then the user code will commit transactions started by the OpenMP runtime, which will violate the synchronization,

and may cause the application to crash.

Similar problems can occur when executing C++ virtual methods, or JIT compiled code, or when writing to uninitialized pointers. We have been able to reproduce the problem in a controlled setting, but we have not seen it occur in the wild. The problem could be avoided by extending OpenMP with clauses which indicate whether synchronization directives can be speculated.

Recently, a lot of effort has gone into proposals for extending OpenMP with TM support. Bae et al. [BCKT14] proposed clauses which specify locking strategies, as hints: The hints include specifying that locks and locks and critical sections should use lock elision if possible. For our purposes, their hints could be extended to all OpenMP directives.

Wong et al. [WAG+14] proposed two new directives: `synchronized`, a transactional version of critical sections, and `transaction`. The `transaction` directive provides transactional execution, with defined semantics for C++ exceptions, for transaction-safe code. Transaction-safe code must follow some mild restrictions, such that it can be executed speculatively by both HTM and STM implementation.

OpenTM [BMT+07] is a programming interface that extends OpenMP with speculative capabilities. They propose (primarily) three new directives that support speculation: `transfor` and `transsection` and `speculation`. They also support nested speculations.

Pyla et al. [PRV11] provides a framework for exploiting coarse-grained parallelism in OpenMP. They introduce regions where speculations take place (*speculative regions*), which can also be nested (speculation within speculation). They propose a directive (`speculate`) to simplify using them.

Miloš et al. [MFU+08, MFG+08] gives an overview *where* in OpenMP speculations can be used. They introduce a new clause, `transaction`, which can be coupled with existing directives to provide speculation. They show how to proposed directives would interact with Nebelung (a STM-based run-time system) and Mercurium [BDG+04] and evaluate their strategy on a set of synthetic benchmarks and a Gauss-Seidel application.

4.4 Concluding remarks

In this chapter, we have shown how to apply HTM to execute some of the most frequently used directives in OpenMP speculatively — minimizing blocking time. We use HTM to speculatively avoid blocking in critical sections, barriers, and taskwaits, transparently to the programmer. Minimizing blocking time can improve execution time in parallel applications that need to synchronize.

Our speculative execution of critical sections handles contention better than Speculative Lock Removal (SLR) without sacrificing scaling. Avoiding blocking of taskwaits and barriers can improve performance by up to 41 % on a micro-benchmark which traditionally blocks most of the time, but it does not significantly improve the performance of our 4 core system on the Barcelona OpenMP Task Suite (BOTS). We expect that future systems with more cores will benefit further from our approach, since they are more likely to have contended critical sections.

Overall, this chapter presents new synchronization techniques applying HTM efficiently — minimizing blocking and contention — without requiring programmer intervention. In the next chapter we will investigate how to write parallel code — requiring programmer intervention — to further optimize for HTM.

Chapter 5

Optimizing for Lock-Elision

In the previous chapter we developed speculative synchronization methods which can reduce execution time and energy consumption. Using a new form of lock-elision — i.e. speculative execution of critical sections — we were able to execute concurrent operations on a shared map data structure with reasonable scalability, while avoiding a traditional downside in lock-elision, the Lemming effect.

In this chapter we attempt to reduce energy consumption and execution time from a different angle: Rather than developing new synchronization techniques that further exploit transactional memory, we present five guidelines for designing code and data layout to minimize the risk of transactional conflicts. We illustrate these guidelines, by applying them to the design of *BT-trees*, a new ordered map. Evaluating BT-trees on standard benchmarks shows that they are up to 5.3 times faster than traditional maps using lock-elision, and up to 3.9 times faster than state of the art concurrent ordered maps.

In this chapter we further illustrate that writing code and data structures specifically for HTM can yield faster and simpler implementations, than writing code and data structures for traditional fine-grained synchronization, by developing a new lock-free map *ELB-trees*, which applies the lessons learned from the BT-tree design. Operations on ELB-trees have weaker semantics than regular maps, to maintain most of the speed and scalability benefits of BT-trees without HTM. The lock-free design appears to scale as well as the transactional design, but

it is not as general or efficient, because it does not have the strong ordering guarantees provided by TM.

5.1 Designing for HTM

In this section we provide general guidelines for designing parallel data structures and algorithms that benefit from lock-elision. The main goal of these guidelines is to illuminate the benefits and pitfalls of HTM, so we can reason about them up front.

There are 3 primary causes of HTM transactions failures:

1. Limitations in the hardware support.
2. Conflicts in the transactions read set — i.e. another thread wrote to the transactions read set.
3. Conflicts in the transactions write set — i.e. another thread read or wrote to the transactions write set.

The hardware limitations cause transactions to fail on false sharing, too large transactions, system calls, and page faults. These limitations can largely be avoided by **(1) optimizing spatial locality**, i.e. packing data tightly, and **(2) avoiding system calls and page faults in the critical section**. Traditional synchronization also benefits from avoiding system calls, page faults, and optimizing spatial locality, but failure to do so is less dramatic: system calls, page faults, and poor spatial locality cause transactions to fail, while with traditional synchronization it just causes cache line evictions and slightly longer critical sections. Even without hardware limitations on HTM, page faults, system calls, and large transactions would still reduce transaction's probability of succeeding, because they increase the sizes of the transaction's read and write sets.

To avoid conflicts in the read set you should, **(3) use data structures and access memory such that the memory which is most frequently written is least frequently read**. Guideline (3) is similar to saying avoid true sharing, which is also a good idea with traditional synchronization, as it will minimize the amount of synchronization. To avoid conflicts in the write set you should, **(4) minimize the time from the first visible write to the transaction's commit**: Try to write as late as possible in the transaction, and prefer copy-on-write to long in-place writes. Guideline (4) is specific to transactional memory which use eager conflict resolution, i.e. detect conflicts in transactions

before they commit. With eager conflict resolution, a transaction can fail from a conflicting write after the write has executed. Delaying the write will reduce the time that the transaction is exposed to such conflicts. Current HTM implementations from Intel and IBM presumably use eager conflict resolution because it can be built on top of existing cache coherency protocols [Lev13, BMV+07].

Our guidelines can also be used to estimate if using HTM will be beneficial: If you can follow the guidelines — i.e. have a very short duration from the first write to commit, avoid writing to frequently read data, and avoid limitations in HTM — then you are likely to benefit, because most transaction should succeed. If most of the transactions succeed, then the size of the critical section is no longer important. As a consequence you should **(5) worry less about the size of critical sections**: lock-elision on a single *coarse-grained* lock can scale well if most transactions succeed. Coarse grained locking simplifies writing parallel code to the point where it is hardly any more difficult than writing sequential code.

Using several locks per data set, or another kind of *fine-grained* synchronization, would not be efficient use of HTM: Starting a transaction with HTM on Intel Haswell processors is 3 times more expensive than acquiring a lock and coarse-grained synchronization with HTM already permits parallelism within critical sections. Fine-grained synchronization will rarely benefit from the increased scalability of HTM, it multiplies the overhead of starting transactions, and it causes new challenges, such as providing safe memory reclamation (See Section 5.4.2). Typical safe memory reclamation solutions imply a hefty performance and space penalty, and may limit which memory locations can be accessed in parallel algorithms. If you really need fine-grained synchronization, then you should use regular locks or atomic operations — not HTM.

To illustrate the relative merits of HTM for coarse grained synchronization, and atomic operations for fine grained synchronization, we have developed a map data structure optimized for HTM in Section 5.2, and a lock-free map data structure optimized for atomic operations in Section 5.4.

5.2 BT-trees

In this section we present the design of BT-trees, a design which is largely driven by the guidelines in Section 5.1. BT-trees are ordered maps supporting the operations SEARCH, INSERT, and REMOVE, for querying, updating, or removing key-value associations. BT-trees are search trees, where the root node represents the entire range of keys, and each of its children represents a smaller subset of

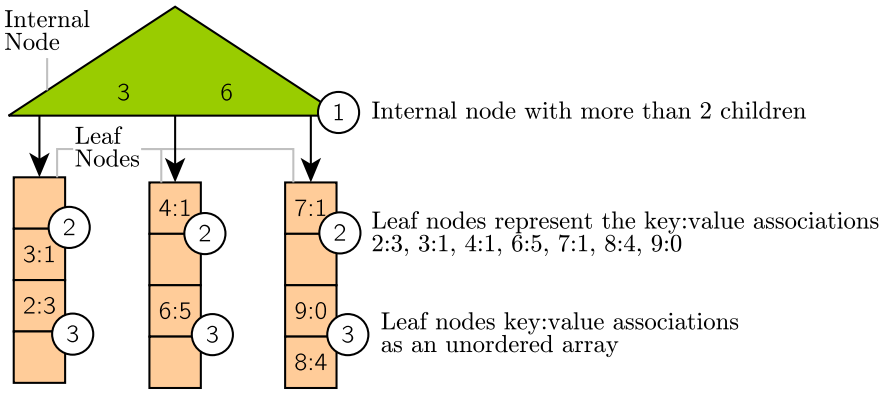


Figure 5.1: Illustration of a BT-tree and BT-trees general features.

the keys. The children are ordered from lowest (child 0) to highest, so that the lowest child can hold the lowest keys, and the highest node can hold the highest keys. The guidelines mostly influence the BT-trees data layout, how BT-trees are balanced, and how they manage memory.

Figure 5.1 illustrates the 3 main features of BT-trees' data layout:

1. Internal nodes can have more than two children and keys. This reduces the height of the tree while improving spatial locality to key references (recall Guideline 1).
2. All key-value associations are stored in leaf nodes. This ensures that most reads are to internal nodes, which rarely change, reducing true sharing (Guideline 3).
3. Leaf nodes store multiple key-value associations in an unordered array. This allows most INSERT and REMOVE operations to only write to the actual key-value association, and only at the very end of transactions (Guideline 4).

BT-trees classify as *external* search trees, and *multiway* search trees because of feature 1 and 2 respectively. As a result of the BT-tree data layout features, all BT-tree operations consist of 3 steps:

1. Finding the leaf node l representing the key.
2. Inspecting l's keys.
3. Balancing l or operating on l based on its keys.

```
1 class E<K, V> {K key; V value; }; // Key-value pairs
2
3 class alignas(64) L<K, V> { // Leaf nodes
4   E<K, V> e[L_C]; // Unordered key-value pairs
5 };
6
7 class alignas(64) I<K> { // Internal nodes
8   I* child[I_C]; // Pointers to children
9   int size; // Number of children
10  K key[I_C - 1]; // Internal node keys
11 };
12
13 class BT { // BT trees
14   int height; // The tree's height
15   I* root; // Pointer to the tree's root
16   Lock lock; // The tree's lock
17 };
```

Listing 5.1: Type definitions for BT-trees.

The operations only write to internal nodes when balancing BT-trees, making it important to reduce the frequency of balancing. BT-trees' operations rarely balance the tree, because of their balancing scheme: unbalanced nodes are balanced in a top down fashion while searching through the tree.

- Full nodes are split into two nodes, and nodes with 2 elements are merged with their siblings;
- If balancing would merge the two only children of the root, we instead replace the root reducing the trees height; and
- If balancing would split the root, we introduce a new root node increasing the trees height.

From an algorithmic point of view, this scheme ensures that all leaf nodes have the same depth, and that all non-root nodes have at least 2 elements, giving a worst case height of $log_2(n/2) + 1 = log_2 n$. Given that the nodes have capacity C, the expected height is $log_{\frac{2+C}{2}} \frac{n}{2} + 1$. The scheme also provides a control knob for the frequency of balancing: increasing C reduces the frequency of node balancing, reducing the frequency of write to internal nodes, and ultimately reducing true sharing and the number of writes.

When balancing BT-trees, we replace the entire parent of the unbalanced nodes — rather than replacing the unbalanced nodes by changing their parent's child pointers. Replacing the parent node means that balancing has to copy the old parent node, which is more expensive, but it reduces the number of writes

to internal nodes. Perhaps more importantly, it ensures that the write to the internal node is the last thing in the transaction, minimizing the time from the first visible write, to the transactions commit, as per Guideline 4.

Memory allocation typically involves both page faults and system calls. BT-trees operations avoid performing system calls and page faults in the critical section (Guideline 2), by preallocating 6 nodes before entering a critical section. Preallocating 6 nodes ensures that no additional memory allocation is needed in the critical section. The allocated nodes are stored on a stack created from the allocated memory, thereby touching the page of the allocated nodes, causing the page fault before the transaction starts.

Listing 5.1 illustrates how BT-trees, key-value pairs, and nodes are represented in pseudocode resembling C++. The classes `I` and `L` represent internal and leaf nodes respectively, while `E` represents key-value pairs. Internal nodes have other internal nodes or leaves as children. Leaf and internal nodes are aligned to cache line boundaries by using the C++11 `alignas` keyword, and allocating with `new`. Each leaf node can store up to L_C key-value pairs, and internal nodes have up to I_C children, where $L_C, I_C \geq 6$. The lower bound node capacities of 6 ensure that we can split a full node into two nodes with at least 3 children or key-value pairs.

We normally use BT-trees with up to 32 children for each internal node, and 32 key-value pairs for each leaf node. When using 64 bit pointers and 32 bit keys and values, this corresponds to 384 bytes for internal nodes and 256 bytes for leaf nodes, or exactly 6 and 4 cache lines respectively.

Listing 5.2 summarizes how the REMOVE operation works. The REMOVE operation first traverses the tree from its root to the leaf node which may hold k, while balancing any unbalanced node on the path (Step 1). Upon arriving at a leaf node, the REMOVE operation balances the leaf if it is unbalanced. Otherwise, the operation iterates over the keys in the leaf node, looking for a match (Step 2). If it finds a match, it returns the keys value and removes from the tree. Otherwise there was no match, and REMOVE returns NO_MATCH (Step 3).

Listing 5.2 is a summary, and it glosses over some technical details. The following section describes how INSERT, SEARCH, and REMOVE actually operate.

```
REMOVE(T, k)
 1  // Step 1: Find the leaf node l representing the key k
 2  repeat
 3       PREALLOC6NODES()
 4       LOCK()
 5       l = FIND-LEAF(T.root, T.height, k)
 6  until l ≠ RESTART
 7  // Step 2: Inspect l's keys (l.key[0 .. L_C-1])
 8  u = 0     // Used key-value pairs
 9  m = −1  // Matching key-value pair index
10  for i = 0 to L_C − 1
11       if l.element[i].key ≠ EMPTY
12            u = u + 1
13       if l.element[i].key == k
14            m = i
15  // Step 3: Balance l or remove k from l
16  if m == −1
17       return NO_MATCH
18  if T.root == l or u > 2
19       v = l.element[m].value
20       l.element[m].key = EMPTY
21       UNLOCK()
22       return v
23  BALANCE(l)
24  UNLOCK()
25  return REMOVE(T, k)

FIND-LEAF(x, h, k)
 1  if h == 0
 2       return x // x is the leaf
 3  if ¬IS-BALANCED(x)
 4       BALANCE(x)
 5       UNLOCK()
 6       return RESTART
 7  for i = 0 to x.size − 2
 8       if x.key[i] ≥ k
 9            return FIND-LEAF(x.child[i], h − 1, k)
10  return FIND-LEAF(x.child[x.size-1], h − 1, k)
```

Listing 5.2: BT-tree remove operation pseudo code.

```
1 bool remove(const K& k, V& res, BT* t) {
2   I* p, **pp; // Parent of leaf node
3   int ci; I* c; // leaf node
4   while (true) {
5     // 1. Find the leaf node
6     findNode(k, pp, p, ci, c, t);
7     // 2. Operate on the leaf node
8     switch(remL(k, (L*) c, res)) {
9     case SUCCESS:
10      release(lock);
11      return true;
12    case FAILURE:
13      release(t->lock);
14      return false;
15    case MERGE: // 3. Merge if near empty
16      mergeL(pp, p, ci, c, t);
17      release(t->lock); break;
18    case SPLIT: // 3. Split if near full
19      splitL(pp, p, ci, c, t);
20      release(t->lock); break;
21    }
22  }
23 }
```

Listing 5.1: Remove operation. Insert and search operations have the same structure

5.2.1 Implementation

In this section we describe how INSERT, SEARCH, and REMOVE actually work, including technical details, such as tracking the parent of the visited node, creating balanced nodes, and some implementation optimizations which improve performance significantly.

All BT-tree operations follow the same template, as illustrated by Listing 5.1:

1. Find the leaf node which may hold the key (Line 6).
2. Perform the operation on the leaf node (Line 8 – 14).
3. If the leaf node is full or almost empty, split or merge it and try again (Line 15 – 22).

Listing 5.2 illustrates how we search, insert, and remove from leaf nodes. BT-trees split full leaf nodes when inserting into them (Line 30), and merge non-root leaf nodes if they only have 3 key-value pairs when removing from them (Line 47 – 48). The operations iterate over up to L_C key-value pairs in very simple loops, which can easily be unrolled manually, or by a compiler. We found that fully

unrolling the loops in `srchL` and `insL` is very beneficial, while `remL` benefits more from specializing the loops in two ways: (1) after the remove operation finds a match it should enter a new loop which only tracks the size of the node, and (2) stop counting the number of used key-value pairs after 3 have been found. We arrived at these implementations by optimizing and stress testing the implementations on generated leaf nodes, in the adaptive development procedure described in Section 2.3.1.

Listing 5.3 illustrates how we find the leaf node which may hold a given key `k`: Preallocate six nodes, begin the critical section, and iteratively traverse from the root to the child which may hold `k` until we reach a leaf node. The internal nodes are traversed by performing a linear search over the keys in the internal nodes (Line 19 and 34). Any node with fewer than 3 children is merged (Line 23 – 27), and any full node is split (Line 13 – 17 and 28 – 31). After balancing nodes, we commit the transaction and restart the function. In order to balance nodes we keep track of the current node `c`, its parent `p`, and the pointer to the parent, `pp`. By only tracking the pointer to the parent, rather than the grandparent, we can simplify balancing nodes which do not have grandparents, specifically balancing of the root node and its children.

Splitting and merging nodes is handled by the same balancing function given different arguments. We split one node to produce two nodes ($in = 1, out = 2$), and we merge two nodes to produce one or two nodes ($in = 2, out = 1 \vee out = 2$). Merging produces one output node when the two input nodes have a combined size less than or equal to $b = \frac{2}{3}(C + 2)$, where C is the capacity of the output node type, that is L_C or I_C. We decide between merging to one or merging to two nodes like this, because it maximizes the number of operations required to bring the new nodes out of balance: if we produce one node, it will take at least $C - b$ operations to fill it, and at least $b - 2$ operations to reduce a merged node's size to 2 — i.e. it will take at least $\min(C - b, b - 2)$ operations to unbalance the new node. if we produce two nodes, it will take at at least $C - \frac{b}{2}$ operations to fill one of them, and at least $\frac{b}{2} - 2$ operations to reduce one of the nodes' sizes to 2 — i.e. it will take at least $\min\left(C - \frac{b}{2}, \frac{b}{2} - 2\right)$ operations to unbalance the new nodes. We minimizing the number of operations it takes to unbalance the new node(s) by solving $\min(C - b, b - 2) = \min\left(C - \frac{b}{2}, \frac{b}{2} - 2\right)$

Listing 5.4 illustrates how we balance internal nodes are balance two internal nodes `c1` and `c2`, with the parent `p` and `c1 == p.c[i] && c2 == p.c[i+1]`. The balanced internal node's children are produced in Line 6 – 9 by copying the first $s = \lceil (c1.size + c2.size)/out \rceil$ children from the unbalanced node(s) to the first balanced node, `n1`, and then copying the remaining children to the second balanced node, `n2`, if $out = 2$. We produce the keys of the balanced internal nodes in Line 10 – 14, retaining the order of the keys in the unbalanced

```
1 Res srchL(const K& k, L* l, V& res) {
2   for(int i = 0; i < L_C; i++) {
3     E<K, V> e = l.e[uf32(i)]; // Look at all keys
4     if (e.k == k) { // If we have a match
5       res = e.v; // Return the matching value
6       return SUCCESS;
7     }
8   }
9   return FAILURE; // No matches in the leaf node
10 }
11
12 Res insL(const K& k, const V& v, L* l) {
13   bool unused = false;
14   int j; // Look at all keys
15   for(int i = 0; i < L_C; i++) {
16     E<K, V> e = l->e[i];
17     if (e.k == 0) {
18       unused = true;
19       j = i; // Remember unused key-value pairs
20     }
21     if (e.k == k) {
22       l->e[i] = {k, v}; // Replace any match
23       return SUCCESS;
24     }
25   }
26   if (unused) {
27     l->e[j] = {k, v}; // Otherwise, replace any
28     return SUCCESS; // empty key-value pair
29   }
30   return SPLIT; // Otherwise split the leaf node
31 }
32
33 Res remL(const K& k,  V& res, L* l) {
34   bool match = false;
35   int m, n = 0; // Look at all keys
36   for(int i = 0; i < L_C; i++) {
37     E<K, V> e = l->e[i];
38     if(e.k != 0) {
39       n++; // Track unused keys
40     }
41     if(e.k == k) {
42       m = i; // Remember matching key
43       res = e.v; // and value
44       match = true;
45     }
46   }
47   if(n <= 2 && !isRoot(l))
48     return MERGE; // Merge nearly empty nodes
49   if(match) {
50     l->e[m].k = 0; // Remove matching key
51     return SUCCESS;
52   }
53   return FAILURE; // No matching key
54 }
```

Listing 5.2: Operations on leaf nodes

```
1 void findNode(K k, I**& pp, I*& p, int& ci,
2     I*& c, BT* t) {
3 start: // Allocate nodes before transactions
4   ensureCapacity(6);
5   pp = &(t->r);
6   p = 0; // The root has no parent
7   acquire(t->lock);
8   c = r; // Start at the root
9   int h = t->h;
10  if (h == 0)
11    return; // The root is a leaf node
12  int size = c->size;
13  if (size == I::C) { // Split the root
14    splitRoot(pp, c, h, size, t);
15    release(t->lock);
16    goto start;
17  }
18  p = c; ci = 0; // Traverse to child
19  while(p->k[ci] <= k && ++ci != size - 1) {}
20  c = p->c[ci];
21  while (--h > 0) {
22    size = c->size;
23    if(size == 2) { // Merge small nodes
24      mergeI(pp, p, ci, c, );
25      release(t->lock);
26      goto start;
27    }
28    if(size == I_C) { // Split full nodes
29      splitInternal(c, p, pp, ci, size, t);
30      release(t->lock); goto start;
31    }
32    pp = &p->c[ci]; // Traverse to child
33    p = c; ci = 0;
34    while(p->k[ci] <= k && ++ci != size - 1) {}
35    c = p->c[ci];
36  }
37 }
```

Listing 5.3: Finding the leaf node which may hold the key k.

nodes, and the key which the parent node used to separate the unbalanced node (`p->k[i]`): First we order `p->k[i]` and the keys from the unbalanced nodes, and then copy the nodes into the balanced nodes such that the first balanced node receives the first $\lceil (c1.size + c2.size)/out \rceil - 1$ keys, and the second balanced node receives the last $\lfloor (c1.size + c2.size)/out \rfloor - 1$ keys. Finally, If $out = 2$ we set the new parent node's `i`'th key to the $\lceil (c1.size + c2.size)/out \rceil - 1$'th key from the ordering (Line 18). The actual implementation uses more complicated code than Listing 5.4 to avoid copying the keys and child pointers into intermediate arrays, but is functionally equivalent.

The key-value pairs of balanced leaf nodes are produced the same way child pointers are produced for balanced internal nodes, except the key-value pairs have to be partially sorted: We partially sort the key-value pairs into two evenly sized and balanced leaf nodes, such that all keys in the first leaf node are smaller than the smallest key in the second leaf node. We implemented the partial sorting with Hoare's quick-select algorithm [Hoa61], after experimenting with Floyd and Rivest's algorithm [FR75], because quick-select showed higher throughput.

5.3 Evaluation of BT-trees

5.3.1 Experiment setup

We evaluate BT-trees, Chromatic trees, and Java `ConcurrentSkipListMap` on the machine described in Table 5.1. Chromatic trees are a state of the art lock-free ordered map, implemented as a relaxed red-black tree, which we acquired from Brown's homepage [Bro]. Java `ConcurrentSkipListMap` is a well established lock-based ordered map, implemented as a skip list. We also evaluate GCC's STL implementation of `map` and `unordered_map` (v4.9.1) where we synchronize using SLR lock-elision. The C++ map implementations all use the

Table 5.1: Experimental machine

Processor	Intel Xeon E3-1276 v3@3.6GHz
Processor specs	4 cores, 8 threads
Processor specs(2)	32KB L1D cache, 8 MB L3 cache
C++ Compiler	GCC 4.9.1
Java Compiler/Runtime	Oracle Server JRE 1.8.0_20
Operating system	Ubuntu Server 14.04.1 LTS
Kernel	3.17.0-031700-generic
libc	eglibc 2.19

```
1 void balanceI(I* c1, I* c2, I* p, I** pp,
2     int i, int in, int out, BT* t) {
3   I* n1,*n2, I* p1; // The new nodes
4   int s = ceiling((c1->size + c2->size) / out);
5   // Fill the new nodes
6   auto ccomb = concatenate(c1->c, c2->c);
7   memcpy(n1->c, ccomb, s * sizeof(void*));
8   memcpy(n2->c, &ccomb[s],
9       (c1->size + c2.size - s) * sizeof(void*));
10  auto kcomb = in == 2 ? c1->k :
11      concatenate(c1->k, p->k[i], c2->k); // Gather the keys
12  memcpy(n1->k, kcomb, (s - 1) * sizeof(K));
13  memcpy(n2->k, &kcomb[s],
14      (c1->size + c2.size - s - 1) * sizeof(K));
15  p1.c[i] = (L*) n1; // Insert the balanced nodes
16  if(out == 2) {
17    p1.c[i + 1] = (L*) n2;
18    p1.k[i] = n2.e[0].k;
19  }
20  int pSize = 0;
21  if(p != 0) {
22    pSize = p->size;
23    ... copy p's other children and keys
24    if(isRoot(p) && pSize == 2) {
25      t->h--; // Merging the root
26    }
27  } else {
28    t->h++; // Splitting the root
29  }
30  p1->size = pSize + out - in;
31  *pp = p1; // Replace the parent
32  dealloc(c1, c2, p);
33 }
```

Listing 5.4: Balancing internal nodes.

memory allocator provided with Intel TBB v4.3_20141023.

We use the experiment from Brown et al. [BER14], and port the experiment to C++ to evaluate the C++ maps. The experiment has been reproduced in several recent papers [NM14, DVY14]. The Java were tested with the test infrastructure hosted on Brown's website.

In the experiment up to 8 threads operate in parallel on one map for 5 seconds, after pre-filling the map with n key-value pairs. After the 5 seconds, we record how many operations the threads completed. The C++ implementations also record several performance metrics, such as cache misses. The operations' keys are 32 bit integers, uniformly sampled from 1 to k, where k is either 100, 10,000, or 1,000,000. We evaluate 3 workloads with different proportions of insert, remove, and search operations:

1. Update, with 50 % insertion, 50 % removal ($n = k/2$);
2. Mixed, with 70 % searches, 20 % insertion, and 10 % removal ($n = 2k/3$); and
3. Constant, with 100 % searches ($n = k$)

Each experiment is run in separate processes, which repeat the trial 50 times. We pre-fill the map with n key-value pairs, because it is the expected number of elements in a map after infinitely many operations.

To minimize any overhead in Java implementations we use an up to date Java Server runtime, which compiles early, and allow the Java virtual machine to consume up to 3 GB memory. By comparison, all of the C++ implementations consumed less than 80 MB memory. It might seem strange to compare the performance of data structures implemented in Java with other data structures implemented in C++, but it is in fact quite commonplace [BER14,NM14,DVY14]. The experimental machine has an Intel Haswell processor which officially only supports HTM "for software development", because the processor has a bug such that "software using the Intel TSX (Transactional Synchronization Extensions) instructions may result in unpredictable system behaviour" [Int14a]. Newer Intel processors officially support HTM: At the time of writing (July 17th, 2015), TSX instructions are supported by all released Broadwell desktop and server processors, by 6 out of 8 released Broadwell embedded processors, and by 6 out of 34 released Broadwell mobile processors [Int15b] (See Appendix B).

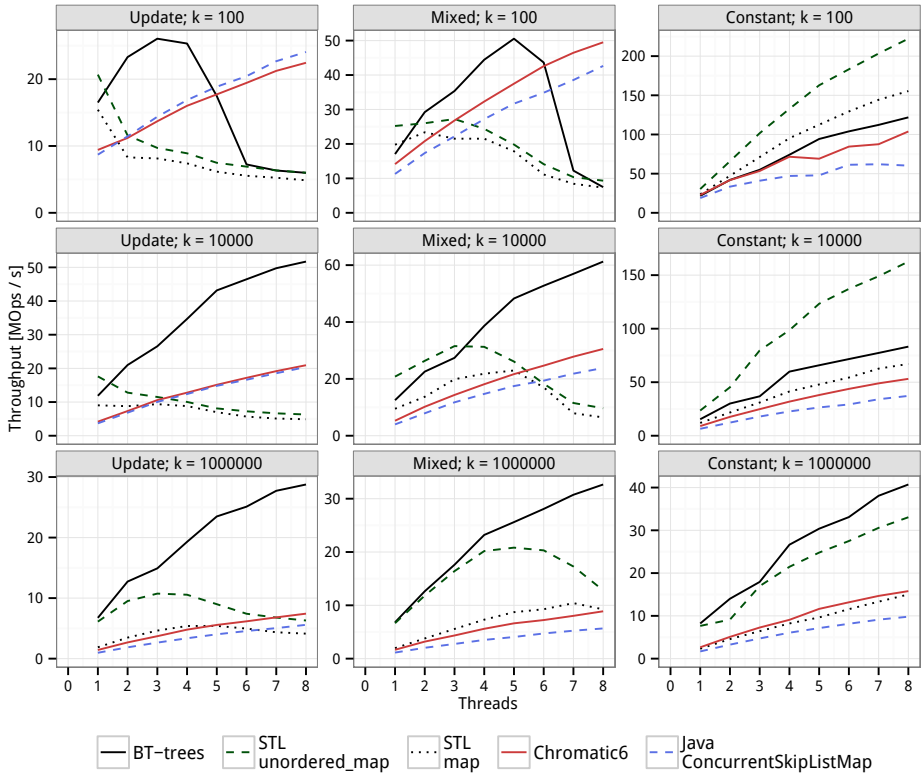


Figure 5.2: Mean map throughput as a function of threads for 3 workloads and 3 key ranges.

5.3.2 Results

Throughput

Figure 5.2 shows the throughput of maps as a function of the number of threads under 9 different workloads and key ranges. The plots are labeled with their workloads and key ranges (k). Figure 5.3 shows the number of L1 cache misses per operation, peak memory consumption, and energy consumption per operation, respectively, as measured by PAPI, version 5.40, `getrusage`, and Intel RAPL. We were only able to measure this data for the C++ map implementations because the measurement interfaces have APIs in C.

BT-trees have the highest peak throughput out of the ordered maps in all workloads. BT-trees advantage is particularly high on large workloads (k = 1,000,000). The traditional concurrent maps are competitive with BT-trees on small workloads ($k = 100$), and even have higher throughput at 8 threads in the

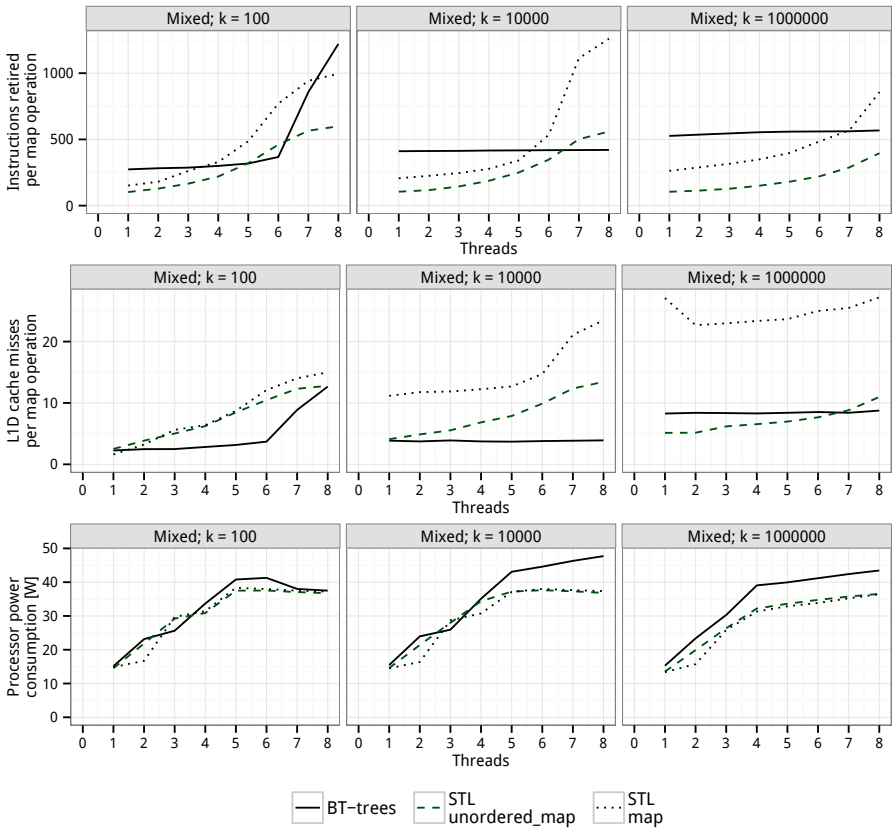


Figure 5.3: Instructions retired, L1 cache misses, and processor power consumption as a functions of the number of threads for the *Mixed* workload on 3 key ranges.

"Mixed; k = 100" and "Update; k = 100" workloads.

BT-trees do not scale well to more than 3 threads in the *Update* and *Mixed* workloads for the smallest key range $k = 100$ because the data structure is highly contended, as can be seen in Figure 5.3. The number of instructions per BT-tree operation increases when scaling beyond 3 threads. The increase is caused by two factors (1) the map operations are retried transactionally, and (2) acquiring the underlying lock executes more instructions when the locks are contended. By comparison, STL maps and unordered maps using lock-elision are contended on all of the *Update* and *Mixed* workloads. For instance the number of instructions executed per STL map operation triples when using 8 threads in the *Mixed* workload with $k = 1{,}000{,}000$ and the results are worse for

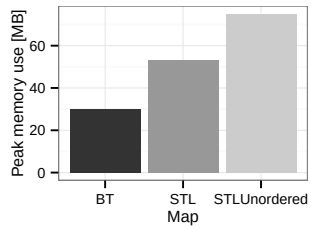


Figure 5.4: Peak memory consumption when $k = 1{,}000{,}000$.

the *Update* workloads and lower values of k.

The lock-elision based maps scale poorly when contended because of the lemming effect [DLM+09]: When operations fail to execute transactionally, they fall back to using the underlying lock. Using the underlying lock increases the risk that the following transactions fail. Once most operations use the lock, their combined throughput performance will decrease below that of single-threaded execution.

When $k > 100$ BT-trees achieve a 2.9-3.4 speedup with 4 threads, and a 4.3-5.6 times speedup with 8 threads. Using more threads does not significantly increase the number of executed instruction or L1 cache misses per BT-tree operation, indicating (1) transactions are rarely retried, and (2) there is little cache contention, hence BT-trees are not contended. We believe the sublinear scaling is caused by two factors: (1) BT-trees yield higher power consumption, which means that they will benefit less from Intel TurboBoost and (2) BT-trees have higher instruction retire rates than the STL-map operations, reducing its benefit from SMT (hyperthreading), since each thread can consume a larger fraction a core's ports.

Compared to STL map operations, BT-tree operations execute 2-3 times as many instructions, but cause 1/4 as many cache misses, in the sequential case. In summary, BT-trees are not significantly affected by contention when $k >= 10{,}000$, but they do not scale linearly when using SMT because of hardware constraints.

Cache performance

BT-trees perform well when $k = 1{,}000{,}000$ compared to Chromatic trees and STL maps, because it benefits from being an external multiway trees, helping its cache performance. Cache performance is more important for larger maps, which cause more cache misses per instruction. Figure 5.3 illustrates the cache performance for the *Mixed* workloads, where STL maps cause 3 times as many L1 cache misses as BT-trees, when $k > 100$. STL maps also cause 6 times as

many L3 cache misses when $k = 1{,}000{,}000$, while the number of L3 cache misses are insignificant for lower k.

STL maps, and binary trees in general, cause more cache misses than multiway trees, because they are higher, and do not fit as well into the cache. Sedgewick [Sed08] observed that the average successful search in left-leaning red-black trees, such as STL maps, traverses $log_2(n)$ nodes, corresponding to 20 nodes in the *Constant* workload when $k = 1{,}000{,}000$. Our results show such operations cause approximately 21 L1 cache misses, which is higher than the expected number of nodes traversed, indicating that traversing red-black tree nodes references more than 1 cache line. By comparison the expected height of BT-trees is $h = log_{17.5}(\frac{n}{2}) + 1$, corresponding to 5.7 nodes in the *Constant* workload when $k = 1{,}000{,}000$, because the expected non-root node has $b = \frac{3+32}{2} = 17.5$ elements. Assuming that traversing BT-tree internal and leaf nodes both cause 2.5 cache line references ($\frac{1+2}{2} + 1 = 2.5$, $\frac{1+2+3+4}{4} = 2.5$), a successful search operation will reference $4.7 \cdot 2.5 + 2 = 13.75$ cache lines. Our results show that a successful search operation on such a BT-tree causes 8 cache line misses. BT-trees have a lower cache line miss to cache line reference ratio than red-black trees because they benefit from hardware prefetching and have better temporal locality. Internal BT-tree nodes change rarely, so they are less likely to be evicted, and more likely to stay cached by the cores.

We expect that Chromatic trees cause approximately as many cache misses as STL maps, but we do not have the infrastructure for fine grained measurement of Java code cache performance. STL maps and Chromatic trees have very similar structures, and as such they have similar sequential throughput, but chromatic trees have far better scalability. They mainly differ in how they are balanced: Chromatic trees are less balanced, and are more expensive to balance, but require less synchronization. Chromatic trees have a constant running time balancing, and the red-black tree property is violated for at most 6 nodes on any path. As a consequence, the sequential performance of the data structures mostly differs for small data sets.

Power consumption

Figure 5.3 shows the processor's power consumption. in the *Mixed* workload. Uncontended maps tend to be more energy efficient when using more threads. In the sequential case, BT-trees are more energy efficient than STL maps when $k >= 10{,}000$, and almost as energy efficient as STL `unordered_maps` when $k = 1{,}000{,}000$. Generally BT-trees are slightly more attractive in the *Update* workload, and slightly less attractive in the *Constant* workload. When using 8 threads and $k > 100$, BT-trees generally consume 25 % more power than STL maps, while still being far more energy efficient. BT-trees' higher power

Table 5.2: Evaluated unordered maps

Data structure name	Details
ConcurrentHashMap [Ora14]	Java (v1.8.0_20)
TrieMap [PBBO12]	Scala-library (v2.11.2)
concurrent_hash_map [Int14c]	Intel TBB (v4.3_20141023)

consumption comes from the cores, while the memory controller power consumption is lower. This is as expected, because operations on BT-trees execute many instructions, but incur few L3 cache misses.

Memory consumption
Figure 5.4 shows the peak memory consumption we measured in the benchmarks. BT-trees' peak memory consumption is approximately 40MB lower than that of STL maps, which is approximately 25 % lower than that of STL `unordered_maps`. When $k < 1{,}000{,}000$ all of the C++ maps have similar peak memory consumption, varying from 6MB to 15MB. We believe the maps have similar peak memory consumptions because most of the memory is consumed by factors other than the maps. Theoretically we would expect the binary trees use at least 32,000,000 bytes to represent 1,000,000 key-value pairs, as representing each key-value pair takes up to 32 bytes: 8 bytes for key-value pair, 16 bytes for child pointers, and 8 bytes for memory allocator data structures and 16 byte alignment. The estimate closely resembles 37 MB, the lowest memory consumption we observed for STL map in the *Constant* workload with $k = 1{,}000{,}000$. We would expect BT-trees to use 17,000,000 bytes to represent 1,000,000 key-value pairs: Every leaf node represents $\frac{3+32}{2} = 17.5$ key-value pairs, and there are approximately 17.5 times as many leaf nodes as internal nodes, giving the estimate $n\frac{\frac{384+16}{17.5}+(256+16)}{17.5} \approx 16{,}800{,}000$. The estimate closely resembles 18MB, the lowest memory consumption we observed in the *Constant* workload with $k = 1{,}000{,}000$.

5.3.3 Comparison with unordered maps

This section evaluates BT-trees in comparison to the unordered maps listed in Table 5.2, on the same experimental setup. The benchmark's design is ideal for hash maps, because the keys have a very dense distribution. A dense key distribution implies that most common integer hash functions are perfect hash functions. In particular, the hash functions of the hash maps in Table 5.2 are perfect hash functions even when truncated to the least significant $log_2(n)$ bits. As a consequence, we expect the hash maps to have lower conflict rates, and

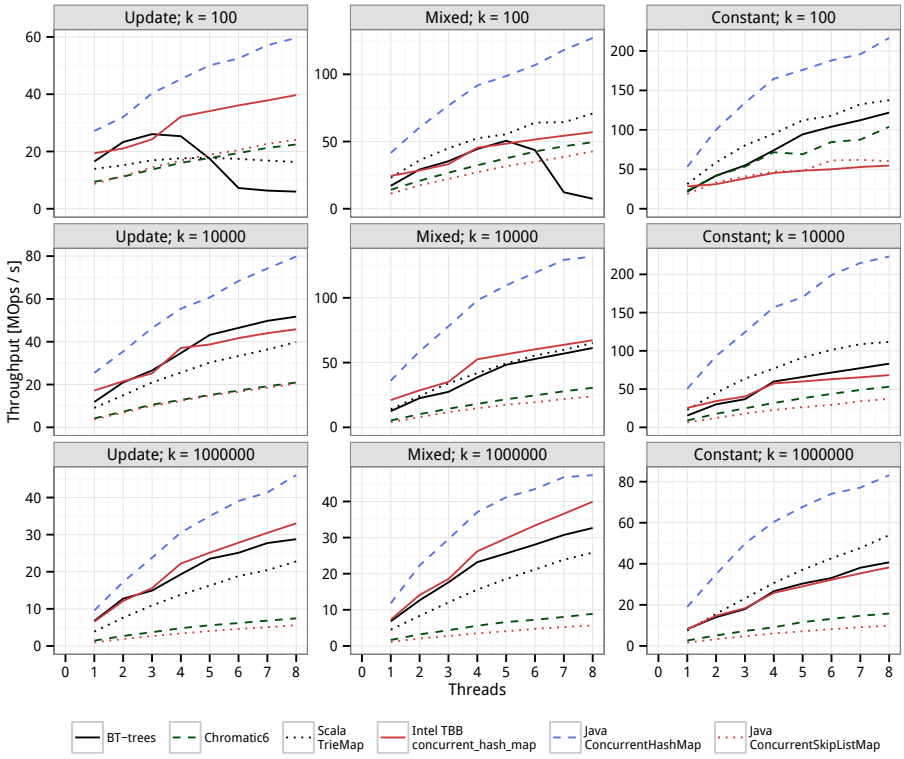


Figure 5.5: Mean map throughput as a function of threads for 3 workloads and 3 key ranges.

higher throughput, than they would have for realistic inputs. Despite being somewhat unrealistic, the benchmark is useful as a stress test, and a best case evaluation for hash maps.

Figure 5.5 shows the throughput of each map implementations on the benchmark. The map implementations single threaded throughput fit the following trend: `ConcurrentHashMap` is always faster than, BT-trees, `TrieMap`, and `concurrent_hash_map`, which are usually faster than `Chromatic6`, which in turn are usually faster than `ConcurrentSkipListMap`. There are 2 deviations from the usual trend: (1) `Chromatic6` are faster than `concurrent_hash_map` in the *Constant* workload when $k = 100$, and (2) `ConcurrentSkipListMap` achieves higher performance than `Chromatic6` in the *Update* workload when $k = 100$. The traditional ordered maps are especially slow compared to BT-trees and the unordered maps on large data structures (large k). The increasing gap is caused by two factors: (1) BT-trees and `TrieMap` being more cache efficient than tra-

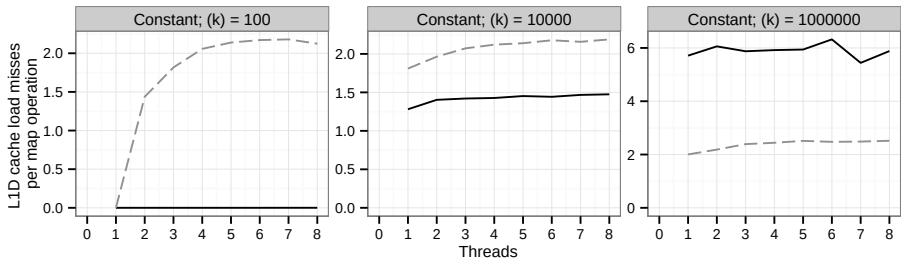


Figure 5.6: The mean number of L1 cache load misses per operations for BT-trees (solid black line) and Intel TBB concurrent_hash_map (dashed grey line).

ditional ordered maps, (2) hash maps have constant asymptotic running time, while skiplists have logarithmic asymptotic running time $O(1)$. To illustrate the performance gap, BT-trees are 1.75, 2.84, and 4.71 times faster than `Chromatic6` in the single threaded *Update* workload for $k = 100$, 10,000, and 1,000,000, respectively. In summary, BT-trees are slower than `ConcurrentHashMap`, similar to `TrieMap`, and `concurrent_hash_map`, and faster than the traditional ordered maps.

The relative performance of the map implementations is similar in parallel cases and the single threaded case. Therefore we will focus on the gray area — the performance of BT-trees when compared to `TrieMap` and `concurrent_hash_map`.

BT-trees compared with `TrieMap`

BT-trees are typically faster than `TrieMap` in the *Update* workloads, except when $k = 100$, and slower in the *Constant* workloads. We believe that this is because the relative cost of insert and remove operations compared to search operations: Insert and remove operations in BT-trees are performed in place on leaf nodes, and have similar costs to searching, while the `TrieMap` insert and remove operations use copy-on-write, which increases their cost relative to search operations.

BT-trees compared with `concurrent_hash_map`

BT-trees and `concurrent_hash_map` have similar performance, except when $k = 100$. BT-trees scale poorly to multiple threads in the *Mixed* and *Update* workloads when $k = 100$, but still achieves higher throughput than `ConcurrentSkipListMap` and `Chromatic6`. BT-trees poor scalability when $k = 100$ is a side effect of using lock-elision; a side effect known as the Lemming effect [DLM+09].

Threads acquire the underlying lock when transactional execution of the critical section fails repeatedly. Acquiring the lock makes concurrent transactions more likely to fail: Once a few transactions fail, many will transactions follow suit.

Meanwhile, `concurrent_hash_map` scales poorly in the *Constant* workloads. When $k = 100$, `concurrent_hash_map` is the slowest map in the *Constant* workload. Figure 5.6 illustrates cache performance of BT-trees and `concurrent_hash_map` in the constant workloads. When going from 1 thread to 8 threads in *Constant* $k = 100$, `concurrent_hash_map` execute more instructions per operation, and cause up to up to 2.3 L1 cache load misses per operation. By comparison no other data structure we measured caused more than 0.01 L1 cache load misses per operation in the *Constant* workload with $k = 100$. The TBB `concurrent_hash_map` scales poorly in this case because it uses a read-write lock per hash bucket. search operations acquire and release read locks by executing a `fetchAndAdd` atomic instruction. The `fetchAndAdd` instructions, as well as any write instructions, invalidates the cache lines of the other cores. By comparison the other maps' search operations do not write to the data structures memory. The TBB `concurrent_hash_map` is not significantly contended for larger values of k, because then the hash map has more buckets, reducing the risk of multiple threads searching adjacent buckets.

5.3.3.1 Summary

In general, BT-trees have excellent space, time and energy performance compared to state of the art concurrent ordered maps, despite using much simpler coarse-grained synchronization. BT-trees also have low memory requirements, and comparable time and energy performance compared to most state of the art concurrent unordered maps, but its performance is not competitive with `ConcurrentHashMap` — the fastest concurrent unordered map we know of.

5.4 ELB-trees

In this section we present the design of ELB-trees, an efficient lock-free map implementation, using a structure similar to BT-trees, but optimized for computer without HTM.

In a fine grained synchronized map, all operations should ideally only synchronize with other operations on the same key. ELB-trees partially achieve that goal: most operations only synchronize on the key-value pair; synchronizing with

```
struct Entry {
  unsigned value  : 32;
  unsigned key    : 31;
  unsigned ro : 1;
}

struct LeafNode {
  Entry e[L_C];
}

struct InternalNode {
  Status status;
  Node* c[I_C];
  Entry s[I_C-1];
  uint64_t size;
  uint64_t height;
}

struct Status {
  Entry key;
  ptr32_t parent;
  int16_t cHeight;
  int16_t type;
};
```

Listing 5.3: C++ code for the data structures for ELB-trees. Leaf nodes store up to L_C unordered entries in the array e. Internal nodes store I_C child pointers in array c and $I_C - 1$ separating entries in array s.

a single CAS operation. Listing 5.3 illustrates the ELB-tree data structures. Internal nodes can have up to I_C children and $I_C - 1$ keys, where $I_C \geq 6$. Leaf nodes store all the key-value associations represented by the tree and the key-value are not ordered within leaf nodes. In many ways the design of ELB-trees mirrors that of the design of BT-trees, described in Section 5.2.

The main differences between ELB-trees' data layout and BT-trees', are differences which enable the coordination balancing without transactions: To coordinate balancing, ELB-trees have an additional `status` field for every Internal node, and an additional `ro` flag for every key-value association. The key-value associations — including the `ro` flag — must be small enough for a CAS operation. ELB-trees' use of `status` fields requires that every non-root node has a grandparent. We ensure that all non-root nodes have a grandparent by adding an internal node whose only child is the tree's actual root — i.e. we add an additional "fake root node" — and by ensuring the tree's actual root is always an internal node. Specifically, when the tree is empty — i.e. it represents no key-value pairs — it will consist of the fake root node, the root node, and one leaf node with no key-value associations.

Operations on ELB-trees typically consist of 3 steps:

1. Find the leaf node l representing the key.
2. Inspect l's keys.
3. Balance l or operate on l based on its keys.

The operations on leaf nodes are similar to those for BT-trees. Listing 5.4 illustrates the remove operation on leaf nodes; the other operations are similar. The

```
1 remove(keyMin, keyMax) {
2 start:
3   entry = {keyMin, 0}
4   // 1. Find "node", the leaf node representing "keyMin"
5   while(!search(entry, 0, &node, &parent, &gParent))
6     rebalance(entry, 0 , &node, &parent, &gParent)
7 retry:
8   // 2. Inspect "node"'s keys
9   smallest = keyMax
10  for(i = 0, used = 0, index = -1; i != L_C; i++)
11    if(node->e[i].ro) // 3. Balance "node"
12      rebalance(entry, 0 , &node, &parent, &gParent)
13      goto start
14    if(node->e[i] != 0)
15      used = used + 1
16      if (keyMin <= node->e[i] < smallest
17        index = i
18        smallest = node->e[i]
19  if (root.c[0]->size != 1 && used <= L_S) // 3. Balance "node"
20    rebalance(entry, 0 , &node, &parent, &gParent)
21  if (index == -1) // 3. Nothing relevant in Leaf node
22    if(successorKey > keyMax)
23      return 0
24    keyMin = successorKey
25    goto start // successorKey is tracked in search
26  // 3. Operate on the key-value pair
27  if(cas(&node->e[index], smallest, 0))
28    return smallest
29  goto retry // Someone removed the smallest fit
30 }
```

Listing 5.4: Pseudo-code for remove operations. It searches for leaf nodes that may have a relevant entry, and attempts to remove one such entry. The start label is used to look for successor nodes, when no relevant entries are found. The retry label is used to look for other candidates for the leaf operation.

```
1  rebalance(key, node, parent, gParent) {
2    // Rebalance the highest unbalanced node
3    while ((root != gParent && parent->size >= KI) ||
4        parent->size <= I_S)
5      if (!search(key, parent->height, &node,
6          &parent, &gParent))
7        return // Someone finished rebalancing
8    helping(STEP1, key, parent->height -1, node,
9      getSibling(node, parent), parent, gParent)
10 }
```

Listing 5.5: Pseudo-code for balancing based on helping. The highlighted lines find the highest unbalanced node.

main difference with respect to BT-trees, is that ELB-trees use CAS operations, rather than transactions, to atomically change key-value associations.

ELB-trees are balanced in a top-down manner, i.e. we balance any unbalanced node encountered while searching the tree. We balance nodes by replacing them with split or merged nodes. Merging produces one output node when the two merged nodes have a combined size less than or equal to $b = \frac{2}{3}(C + 2)$, where C is the capacity of the output node type, that is L_C or I_C. We split lnternal nodes in ELB-trees when they have I_C children, similarly to BT-trees, but unlike BT-trees we split ELB-tree leaf nodes that have more than L_D key-value associations, where $4 < L_D < L_C$. We allow splitting leaf nodes which are not entirely full, because balancing nodes is no longer atomic: After encountering a full leaf node, and deciding to balance the node, another thread could remove a key-value association from it, which means it is no longer full. If we did not split nearly full leaf nodes, then splitting may become a rare occurrence.

The two main challenges the ELB-tree design are:

1. Balancing nodes in a lock-free manner with regular CAS operations
   - CAS operations only change a single field atomically, while balancing updates multiple nodes.
2. Safely accessing nodes which other threads can deallocate at any time.

5.4.1 Balancing

This section describes the ELB-trees synchronization scheme when balancing. We create the balanced nodes for ELB-trees the same way we create balanced

```
1  helping(op, key, cHeight, node, sibling, parent, gParent) {
2  start:
3    switch (op)
4    case NOP:
5      return
6    case STEP1: // Set gParent status field
7      ... get hazard pointer to sibling
8      cas(&gParent->status, 0,
9        {key, parent, cHeight, STEP2})
10     if(gParent->status != {key, parent, cHeight, STEP2})
11       oldStatus = gParent->status
12       break // Help gParent
13     oldStatus = {key, parent, cHeight, STEP2}
14     if (!gParent->hasChild(parent))
15       cas(&gParent->status, oldStatus, 0)
16       return // Someone rebalanced the parent
17     // Otherwise continue
18   case STEP2: // Set parent status field
19     cas(&parent->status, 0, {key, parent, cHeight, STEP3})
20     if (parent->status != {key, parent, cHeight, STEP3})
21       oldStatus = parent->status
22       break // Help parent
23   case STEP3: // Set unbalanced nodes status field
24     if (cHeight == 0)
25       ... rebalance leaf nodes, no more helping
26       cas(&gParent->status, oldStatus, 0)
27       return
28     cas(&node->status, 0, {key, parent, cHeight, STEP4})
29     if (node->status != {key, parent, cHeight, STEP4})
30       oldStatus = node->status
31       break // Help node
32   case STEP4: // Set the nodes siblings status field
33     if (node->size >= KI - 1)
34       ... split the node
35       return
36     cas(&sibling->status, 0, {key, parent, cHeight, STEP5})
37     if(sibling->status != {key, parent, cHeight, STEP5})
38       oldStatus = sibling->status
39       break // Help sibling
40   case STEP5:
41     ... even out the entries in the node and its sibling
42     cas(&gParent->status, oldStatus, 0)
43     return
44   // Find and help the preventing operation
45   ... find nodes involved in operation of oldStatus,
46   ... updating node, sibling, parent, and gParent
47
48   if (gParent->status.matches(oldStatus))
49     return // Someone finished the operation
50   if (parent != oldStatus.parent)
51     cas(&gParent->status,
52       {key, parent, cHeight, STEP2}, 0)
53     return // Someone rebalanced the parent
54   op = oldStatus.type
55   key = oldStatus.key
56   oldStatus = {key, parent, cHeight, STEP2}
57   ... get hazard pointer to sibling, return if unable
58   goto start
59 }
```

Listing 5.6: Pseudo-code for helping balancing. The highlighted lines implement recursive helping and can be ignored on the first reading.

nodes for BT-trees (See Section 5.2.1).

We balance nodes in a lock-free manner as illustrated by Listing 5.5 and 5.6. Listing 5.5 starts the balancing, by identifying the highest unbalanced nodes, and Listing 5.6 balances the nodes in such a way that any other thread can help finish the balancing. We first prevent concurrent modification by making the the node, its parent, and its grandparent read-only — preventing concurrent changes to the nodes — before actually balancing the node. We make Leaf nodes read-only by setting each of their key-value associations' `ro` fields. We make Internal nodes read-only by setting their `status` fields with CAS operations.

We make the nodes read-only in the following order: The grand parent (Listing 5.6 line 6–17), the parent (Line 18–22), the node (Line 23–31), and, if we are merging, the node's sibling (Line 32–39). The parent node can disappear from the tree while we a balancing, because of concurrent balancing, in which case we clear the grand parent's `status` field (Line 15). If we successfully set all the internal nodes' `status` fields, then we balance the nodes (Line 25 and 41) and clear the grand parent's `status` field. If we fail to set any internal node's `status` field then we have to help finish the balancing operation on that internal node. The `status` fields encode which nodes are being balanced, allowing any thread to help finish balancing (Line 45–57). Leaf nodes do not have `status` fields, but their parents do, allowing any thread to help finish balancing. Allowing any thread to finish the balancing operations, whether they started them or not, is important for showing that the operations are lock-free.

5.4.2 Safely accessing nodes

We use a mechanism known as *hazard pointers* [Mic04] to safely access nodes which can be deallocated at any time. Threads must acquire hazard pointers to nodes before accessing them. A thread acquires a hazard pointer by publishing that they are going to access the node, and then checking that the node is still *reachable* from the root node. We acquire hazard pointers while searching through the tree, as illustrated by the highlighted lines in Listing 5.7, and when merging nodes (see Listing 5.6 Line 7 and 57). The node is reachable if its parent points to it, and both the parent node and its children are not being balanced, which we can tell from the `status` field. If the parent's `status` field indicates that it, or one of its children, is being balanced, then the thread must help finish the balancing, and restart the threads operation. Threads must execute full memory barriers after publishes that it will access a node, since most platforms, including AMD64, do not guarantee memory ordering for independent memory reads and writes. In other words, threads have to execute a memory barrier for each node they visit, which is quite expensive.

```
1  search(key, height, *node, *parent, *gParent) {
2  retry:
3    *gParent = &root // The fake root is permanent
4    *parent = root.c[0]
5    hp[0] = *parent // Acquire HP to real root
6    memoryBarrier()
7    if (!*gParent->hasChild(*parent))
8      goto retry
9    *node = *parent->findChild(key)
10   ... Keep track of successor/predecessor
11   hp[1] = *node // Acquire HP to first child
12   memoryBarrier()
13   if (!*gParent->hasChild(*parent) ||
14     !*parent->hasChild(*node))
15     goto retry
16   i = 1
17   while (*parent->height - 1 > height)
18     *gParent = *parent
19     *parent = *node
20     *node = parent->findChild(key)
21     ... Keep track of successor/predecessor
22     i = i == 2 ? 0 : i + 1
23     hp[i] = node // Acquire HP to child
24     memoryBarrier()
25     if (!*gParent->hasChild(*parent) ||
26       !*parent->hasChild(node) ||
27       !*gParent.status.matches(*parent.status))
28       goto retry
29     if (*parent.status.type >= STEP4)
30       return false
31   return true
32 }
```

Listing 5.7: Pseudo-code for the search function. The highlighted lines are not neccessary on your first reading; they check whether the nodes are reachable using hazard pointers.

Each thread can publish up to live 4 hazard pointers: the current node, its parent, and its grandparent, all acquired in the tree search function, and a reference to the sibling of an unbalanced node, being acquired when balancing, for the purpose of merging. Before deallocating nodes, threads must remove the node from the tree, and check that no other thread has published that they have access to the node. Each thread can defer deallocation of up to $8P$ nodes — where P is the number of threads — by storing the node pointer in a thread local array. When the array is full, the thread deallocates nodes which are not in use, by comparing against a hashmap of nodes used by the other threads. The only use of hazard pointers which is specific to ELB-trees is the check for whether nodes are reachable.

5.4.3 Extensions

We have extended ELB-trees to support priority queue and multimap operations. Priority queues require an operation which removes the key-value association with the lowest key, which we implement by searching for a key-value pair with the key 0, and removing the lowest key-value pair we find.

Multimap operations require the capacity to insert multiple keys with different values, which we support by extending the internal nodes' `key` fields into full key-value associations, and directing internal node search with both the key and value. Remove and search operations initially search for a given key, and the value 0, while tracking the lowest key-value association which can be stored in the successor to the leaf node. If the operations initially only find key-value pairs with lower keys than desired, they restart their operation with the lowest key-value pair which can be stored in the successor to the leaf node.

5.4.4 Semantics

ELB-trees contain an initially empty set of entries E_r. ELB-trees offer 3 main operations:

- Search(e_1, e_2) returns e from E_r satisfying $e_1 \leq e \leq e_2$, if such an entry exists. Otherwise it returns 0.
- Remove(e_1, e_2) removes and returns e from E_r satisfying $e_1 \leq e \leq e_2$, if such an entry. Otherwise it returns 0.
- Insert(e) adds e to E_r, if e was not in E_r before. If e was E_r before the behavior is undefined.

The restriction on insert operations, means that it is the programmers responsibility to ensure that there are no duplicate entries. This is given if the entry is unique, for instance if the value is a memory address. The operations cannot generally be expressed as atomic operations; rather they occur over a time interval. Specifically the operations may not read entries that are not in E_r at all times during the operation. As a consequence, series of concurrent operations cannot generally be expressed as occurring serially, that is the semantics are not linearizable.

Appendix A proves the operations' semantics. The following paragraphs presents a a sketch of the proof: Search operations consists of a tree search through internal nodes and linear search of leaf nodes. Internal node search can be reduced to regular tree search through 3 invariants:

1. Rebalancing does not change E_r, and produces the same balanced nodes, regardless data races.
2. The permitted range of entries in nodes is permanent.
3. The search tree invariant ($s[i-1] < v \leq s[i]$) is maintained for all *reachable* nodes.

Due to memory barriers, the linear search through leaf nodes must read every entry in the node for the duration of the operation, ensuring that semantics hold for leaf nodes as well. The correctness of remove and insert operations follow from the semantics of searches. Remove and insert can be summarized as search operations followed by a value based *CAS* operation. The operations continue until they successfully write to the entry. *CAS* operation can only succeed when writing to entries in *reachable* nodes, as old nodes that are no longer reachable, must be read-only.

Lock-freedom is proven by showing that operations eventually restart or complete, and some operation has made progress whenever a node is rebalanced, or any part of an operation is restarted.

5.5 Evaluation of ELB-trees

We use two different PCs to evaluate ELB-trees. Some key parameters are summarized in Table 5.3. Both PCs support *CAS* operations of up to 128 bits, and both run Linux; PC_A runs Debian 6.0.6 with kernel 2.6.38.6, while PC_B runs Scientific Linux 6.1 with kernel 2.6.32. We

Table 5.3: Platforms used for the experiments. PC_A supports hyper-threading.

	Microprocessor and speed		Cores	L1 Size	L3 Size
PC_A	2x Intel Xeon X5570	2.933 GHz	2x 4	8x 32 KB	2x 8 MB
PC_B	2x AMD Opteron	1.9 GHz	2x 12	24x 64 KB	4x 6 MB

Table 5.4: Mean sequential running times in seconds.

Experiment	Platforms					
	PC_A			PC_B		
	$n = 10^4$	$n = 10^5$	$n = 10^6$	$n = 10^4$	$n = 10^5$	$n = 10^6$
ELB-wide multimap	0.0064	0.0717	0.8386	0.0113	0.1231	1.6070
STL-multimap	0.0013	0.0295	0.5737	0.0020	0.0539	1.0952
ELB-multimap	0.0019	0.0243	0.3910	0.0026	0.0343	0.6446
STL-priority queue	0.0006	0.0060	0.6100	0.0008	0.0086	0.0909
ELB-priority queue	0.0016	0.0194	0.2502	0.0020	0.0250	0.4429

compile with GCC (Ubuntu/linaro 4.6.1-9-ubuntu3) using the flags: `-m64 -std=gnu++0x -Ofast -flto -fwhole-program -fno-align-functions -fno-align-labels -fno-align-loops -fno-align-jumps -s -DNDEBUG -fopenmp`.

We run with up to 16 threads, in some cases 24 threads, and preallocate all memory in all experiments to avoid the overhead of memory allocations from influencing the results. Each data point is the average of 160 runs and we present 95 % confidence intervals. Table 5.4 shows the mean sequential running time for all the experiments.

5.5.1 Wide multimap

To emulate use in data bases, we evaluate the performance of ELB-trees when used as wide dictionaries. B-tree variants are often used in data bases, where they have large nodes, suitable for storage on hard drives. The experiment is laid out as follows: p threads each perform n/p operations on a multimap with n entries. 20 % of the operations are insertions, 20 % are removals, and 60 % are searches. The keys used for the operations and the initial entries are sampled from the discrete key distribution $U(1, 2^{\lceil 1+log_2(N) \rceil})$. Every node takes up 4096

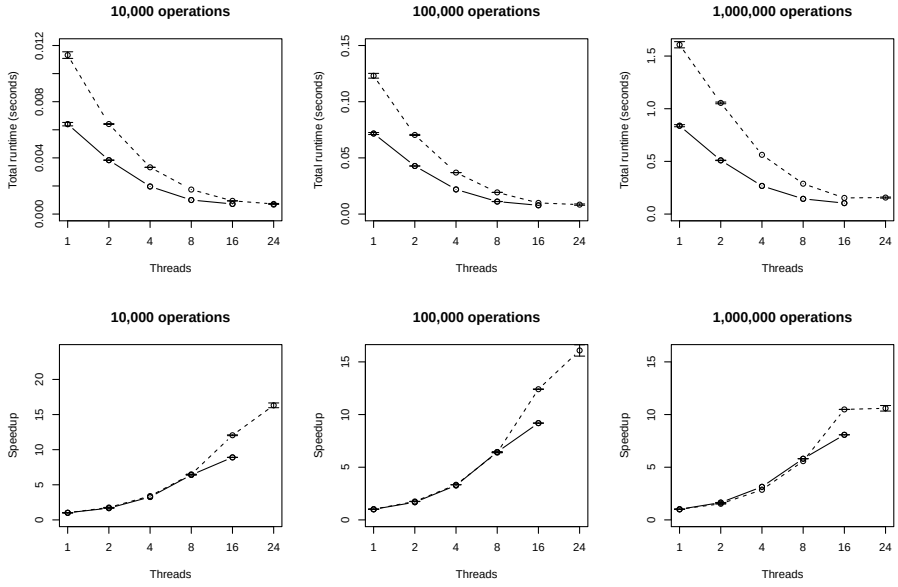


Figure 5.7: Wide multimap runtime and speedup of using n threads relative to 1 thread. Solid line is from PC_A, dashed is PC_B.

bytes. For our implementation, this corresponds to internal nodes with 252 child pointers and leaf nodes with 512 entries. Fig. 5.7 shows the results with $n = 10^4$, $n = 10^5$, and $n = 10^6$, corresponding to tree heights of 1, 2, and 2, respectively. The relative speedup is at most 8.1 for PC_A, and 16.1 for PC_B.

Contention prevents ELB-trees from achieving linear speedup on PC_B. When multiple threads attempt to change the same entry, only one of them can succeed [PK11, Bon12]. To properly synchronize when modifying a value, the cache line of the variable is evicted from other processors. This effect can be seen as observed as a 55 % increase in the number of L2 data cache misses when increasing the number of threads from 1 to 16 on PC_B. Contention is less of an issue for large n, as threads are less likely to modify the same nodes.

Rebalancing introduces serialization of threads, because even with helping all the threads perform the same operation. This affects performance and is an issue in all the experiments. The impact is more significant for larger values of n. This is because the leaf nodes contain more entries, resulting in more time spent in the rebalance operation.

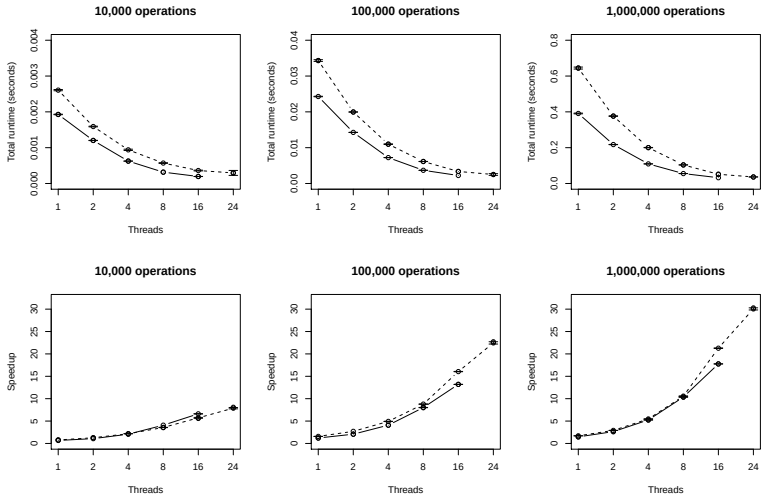


Figure 5.8: multimap runtime and speedup relative to STL multimap. Solid line is from PC_A, dashed is PC_B.

5.5.2 Multimap

Next, we evaluate the performance of ELB-trees, when using node sizes optimized storage in RAM, rather than on hard drives. The only notable difference is that nodes have 32 entries or child pointers, and are considered sparse when containing 4 or fewer entries or child pointers. Leaf nodes are considered dense when they have more than 26 entries. Allowing nodes this sparse may seem space inefficient, but we found that the ELB-trees used significantly less memory than the competing data structure, similar to our findings for BT-trees. This is due to the multiway search structure, which reduces the number of nodes and pointers in the tree.

We compare this with the multimap from GCC's STL implementation libstc++-v3. Fig. 5.8 displays the results of this experiment with $n = 10^4$, $n = 10^5$, and $n = 10^6$, corresponding to tree heights of 3, 3, and 4, respectively. When compared to the single-threaded case, PC_A achieves a peak speedup of 9.9, 10.9, and 12.1 for $n = 10^4$, $n = 10^5$, and 10^6, respectively. On PC_B the figures for the same values of n are 8.8, 13.7, and 17.6. ELB-trees is up to 29.9 times faster than the STL multimap on PC_B.

ELB-trees are approximately 25 % slower in the single-threaded case than the

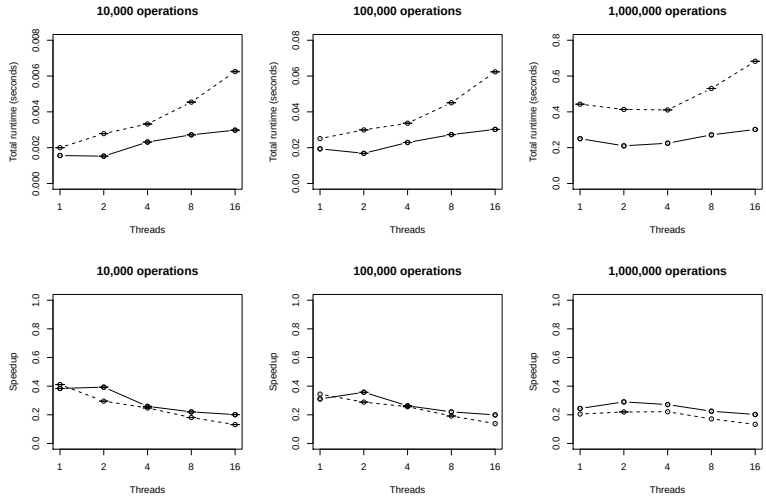


Figure 5.9: Priority queue runtime and speedup relative to STL multimap. Solid line is from PC_A, dashed is PC_B.

multimap for $n = 10^4$, but roughly 60 % faster for $n = 10^6$. The performance at $n = 10^4$ can be explained from the overhead of atomic operations, and ELB-trees executing 5.9 times as many instructions per operation. The STL-dictionaries cause 6.9 times as many L1 cache misses as ELB-trees, which helps explain why ELB-trees are only 25 % slower. The improved performance in the $n = 10^6$ case is due to ELB-trees only executing 3.4 times as many instructions, and the STL-dictionaries causing 5 times as many TLB misses. The difference in the number of executed instructions can partially be explained by operations on ELB-trees having to search through unordered leaf nodes. The cost of such a search is independent of the number of entries in the tree, and is therefore most noticeable for small trees.

5.5.3 Priority queue

Finally, we evaluate the performance of ELB-trees when used as a priority queue. We conduct the following experiment: p threads each perform n/p operations on a multimap with n entries. Half of the operations are insertions, and the other half removes the smallest entry. The keys used for the operations and the initial entries are sampled from the discrete key distribution $U(1, 2^{\lceil 1+log_2(n) \rceil})$. We compare this with the compiler's default STL priority_queue.

We found that using nodes with 16 entries or child pointers was the most efficient. ELB-trees are 60-80 % slower in the single threaded case than the priority_queue, and ELB-trees does not achieve any speedup relative to the STL priority_queue, see Fig. 5.9.

GCC's priority_queue makes use of highly optimized heaps [TDK]. Heaps make excellent priority queues, whereas ELB-trees are primarily designed for dictionaries although they can be used as priority queues. Hence, the poor performance when compared to STL's priority_queue is not necessarily surprising.

In the experiment, contention is high and there is little opportunity for parallelism. Insertions can proceed in parallel, like in the multimap experiments, but all removals attempt to modify the same entry. Few versatile lock-free priority queues exist, and so ELB-trees, even with low performance, constitute an important design point.

5.6 Related work

In this section we will review some of the prior work studying transactional memory and map data structures. These are two huge fields, so unfortunately we have to omit omit many interesting works.

Studies of transactional memory
Transactional memory has been quite a hot topic in computer science since its introduction in 1993 [HM93]. Most of the research has focused either on the programming model for transactional memory, or its implementation, either in hardware or software [CMF+13, DLM+09, DFGG11, ST97, Fra04]. There have also been a several studies of the impact on software which applies transactional memory. Recently, empirical studies confirmed that transactional memory is easier to use than traditional fine-grained synchronization methods [RHW10, PAT11], but not necessarily simpler than coarse grained synchronization.

Our work differs from prior work on transactional memory, by focusing on how to design software, in order to apply HTM efficiently. Our general approach is to apply simple coarse grained synchronization based on SLR lock-elision [Lev13] and redesign data layout to avoid limitations in HTM. Bobba et al. conducted a very related study; a study of performance pathologies for different simulated HTM implementations [BMV+07]. They found that different implementations of HTM will pathologically degrade performance under different workloads. Transactional memory implementations with requester-wins policies, sim-

ilar to the current Intel and IBM implementations, showed a property they called *Friendly Fire*: transactions which conflict with other transactions are likely to fail themselves. We use HTM in the form of lock-elision, which suffers from the Lemming Effect, giving poor performance in the same situation [DLM+09].

Map data structures

Maps, concurrent or otherwise, have been studied for decades, so we have to omit many interesting maps, and mostly focus on search trees.

Bayer et al. [BM70] introduced *B-trees*, which are balanced multiway search trees. B-trees are balanced by splitting nodes when they are full and merging nodes when they are half empty. Splitting and merging nodes may recursively split or merge their parents. B-trees have spawned many related search trees, including *B+trees*, which are external trees, *B* trees* which have stricter balancing guarantees, and recently *B-slack trees*, which have even stricter balancing guarantees than B* trees.

B+trees have been adapted into parallel versions, such as the *PO-B+trees* by Mond et al. [MR85]. PO-B+trees use fine-grained synchronization, with a read-write lock for every node in the tree. Operations on PO-B+trees only hold the lock of the node they are accessing and the node's parent. They avoid holding more locks by relaxing the balancing requirements of B+trees, and instead merge and split nodes while searching.

Braginsky et al. presented a lock-free B+tree, also using relaxed balancing [BP12]. The tree is lock-free because it provides progress guarantees, which cannot be provided when using locks. Lock-free B+trees use atomic operations on individual memory locations instead. They represent node contents with linked lists, and coordinate operations through per-node fields, which track the operations progress.

The idea of coordinating large *lock-free* operations on trees through per-node fields was popularized by Ellen et al., with their non-blocking binary search tree [EFRvB10]. This paper led directly to the *non-blocking k-ary search tree*, a multiway search tree, and the *lock-free chromatic trees* by Brown et al. [BH11, BER14]. *Chromatic search trees* were originally presented by Nurmi et al. [NSS91] as a highly parallel variation of red-black trees with relaxed balancing. The original chromatic trees use a separate thread for balancing to avoid balancing in normal operations, simplifying synchronization. *SF-trees* [CGR12] take this one step further, using a separate thread to balance and remove nodes, to minimize conflicts when using software transactional memory.

Red-black trees and their balancing scheme were introduced by Bayer [Bay72] as an optimized special case of B-trees with 4 children. Each node in a B-tree corresponds to one black and up to two red nodes in a red-black tree. The *left-leaning red-black trees* by Sedgewick [Sed08] restructure and simplify the balancing of red-black trees. Left-leaning red-black trees are used in GCC's implementation of STL maps.

Pugh presented *skip lists* as alternative ordered maps [Pug90]. Skip lists are linked lists with additional *skip links* which are used to skip over parts of the list. The skip links are updated pseudo randomly, maintaining an expected $O(log(n))$ node depth with simple synchronization. Fraser [Fra04], Fomitchev et al. [FR04], and Sundell et al. [ST04] independently developed lock-free skip lists with map operations. Recent studies have found that skip lists scale worse than search trees [BER14].

Briandais and Fredkin independently presented *tries* [DLB59, Fre60], which are trees, but do not compare node keys when traversing. Trie operations are less computationally expensive than balanced search tree operations, but lead to less balanced trees. Litwin et al. [Lit84] presented *hash tries*, where children are indexed based on key hashes, and later presented concurrent implementations [LSV89]. Prokopec et al. [PBBO12] presented *Ctries*, a lock-free multiway hash trie, inspired by the multiway hash trie of Bagwell [Bag00]. The hash tries digest keys to 32 bit integers, so they are only ordered maps if the key is an integer and the hash function is the identity function.

Hash maps are a very popular unordered map implementation. They are typically implemented as an array of linked lists, where the array is indexed by a hash on the key. As the map grows, operations periodically resize by moving the map's contents into new linked lists and arrays keeping the linked lists short. Operations that do not resize are mostly independent, which has enabled efficient concurrent implementations, such as the lock-free hash maps by Maged [Mic02] and by Shalev et al. [SS06]. Mainstream frameworks, such as the Java runtime and Intel TBB, offer efficient lock-based hash-maps [Ora14, Int14c].

The balancing scheme of ELB-trees and BT-trees can be seen as a simplification of that used by B+trees, or a further relaxation of PO-B+trees. BT-trees allow internal nodes to have fewer children than PO-B+trees to further reduce balancing. The attempt to reduce balancing is seen in other concurrent data structures such as chromatic trees, skip lists, Ctries, and concurrent hash maps. SF-trees also try to reduce balancing and optimize for transactions, but with different means for avoiding conflicts: BT-trees and ELB-trees reduce the frequency of balancing, whereas SF-trees defer balancing to another thread; SF-trees defer removing nodes from the tree to avoid conflicts near the root, while we use multiway external search trees, making writes near the root infrequent. The

main similarity of BT-trees and SF-trees is that they both use individual transactions for balancing, rather than allowing an operation and balancing in the same transaction.

5.7 Concluding remarks

Modern multi core processors offer increasing parallel power, but writing efficient parallel code is still difficult. In this chapter we illustrated a simple way of writing efficient parallel code applying hardware transactional memory: Reason about how the code affects the synchronization, rather than custom tailoring new synchronization schemes.

We presented 5 guidelines that can help detect scalability pitfalls, and applied the guidelines to the design and implementation of BT-trees, a new ordered map. BT-trees are 3 times faster and twice as space efficient as state of the art concurrent ordered maps. Unlike other state of the art ordered maps, BT-trees use very simple synchronization. Using the same synchronization on traditional maps — which were not designed according to our guidelines — results in massive synchronization contention, and limited scalability.

We also used the lessons learned from BT-trees to design and implement ELB-trees, a lock-free data structure, which does not require hardware transactional memory. When used as a multimap, ELB-trees scales well with up to 24 processors, but its sequential performance is lower than BT-trees. ELB-trees are slower in the sequential case because their operations have to use safe memory reclamation, which incurs a heavy penaly on every node accessed. The more serious disadvantage of ELB-trees, is that their operations semantics are relaxed and non-linearizable, which make them unsuitable for some use cases.

Chapter 6

Conclusion

In this dissertation we have explored how computer performance, both in terms of energy and execution time, can be improved by adaptively optimizing applications. We initially claimed the hypothesis:

> Middleware adapted to current needs can improve computational performance without being prohibitively expensive to deploy.

In the following sections, we will illustrate how our developments support this hypothesis — summarizing the contributions of this dissertation — before suggesting fruitful avenues for future work.

6.1 Contributions

Our main contributions come in the form of reusable software middleware, libraries, and programming guidelines, ranging from easily deployable and application agnostic, to application specific. The following paragraphs review the contributions in the same order we presented them in the thesis:

`pappeadapt`, a workload aware power governor. A power governor which trades longer execution times for lower power consumption, when doing so reduces the energy consumption. The technique saves energy by adapting the processor's voltage and frequency settings to accommodate the current workload, without requiring any code changes or recompilation. `pappeadapt` detects whether the current workload will benefit from a lower CPU frequency by continually experimenting with CPU frequencies and predicting the performance at all possible CPU frequencies. The predictions are derived from a model based on Amdahl's law, which can accurately predict performance at different CPU frequencies — an attractive property which could also be applied to select processor models or frequencies capable of satisfying a fixed workload's throughput requirements. The execution time prediction model improves on those used in prior power governors, by not assuming that the workloads' performance scaling is a function of the number of cache misses it causes — an assumption which we show does not hold in general.

Speculative execution of OpenMP directives. A group of synchronization techniques which use hardware transactional memory to synchronize while minimizing blocking and the weaknesses inherent in hardware transactional memory, while only requiring that the application is linked against our OpenMP library — permitting further parallelism with little effort, potentially increasing power consumption but decreasing execution time sufficiently to reduce energy consumption. The techniques expand on prior works, by handling contention better and only requiring the subset of transaction memory capabilities offered by current Intel Haswell and IBM Power 8 processors.

Optimization guidelines for hardware transactional memory. An analysis of the properties of current hardware transactional memory implementations, yielding five guidelines for writing code which applies hardware transactional memory efficiently — permitting further parallelism with some programmer effort.

Concurrent data structures. BT-trees are a concurrent maps which apply our guidelines for hardware transactional memory to provide a highly scalable and efficient ordered map implementation — often increasing power consumption, but reducing execution time sufficiently that we can achieve significant energy savings.

ELB-trees are lock-free relaxed maps, which apply the lessons we learned from BT-trees to a concurrent map without requiring hardware transactional mem-

ory — offering the same highly scalable performance as BT-trees, but at a lower single threaded performance. ELB-trees are more complicated than BT-trees — and have a lower single threaded performance — because ELB-trees cannot rely on the strong ordering guarantees provided by transactional memory. Instead ELB-trees struggle with the complexities of safe memory reclamation — safely deallocating tree nodes which may be accessed concurrently — and providing progress guarantees, issues which are not present for BT-trees. The dual development of BT-trees and ELB-trees also provides a thorough illustration of the potential advantages and disadvantages of using hardware transactional memory versus using atomic operations for synchronization.

6.1.1 Discussion

When viewed as a whole, we have shown that reusable middleware — in the form of libraries and power governors — can dramatically improve computational performance, both in terms of energy and execution time efficiency. Our developments build on top of prior techniques, adapting them to current hardware capabilities and requirements, providing great benefits while arguably requiring little deployment effort. We argue that our middleware require little deployment effort, as they only require that one runs our power governor, links against our library, considers the implications of transactional memory, or uses our data structures. While we have by no means solved the problems posed by the end of Dennard's scaling, we believe that our contributions are a clear sign that:

> Middleware adapted to current needs can improve computational performance without being prohibitively expensive to deploy.

The middleware approach to combatting the end of Dennard's scaling is highly attractive because it has a low deployment effort. The low deployment effort means that the middleware can be deployed in realistic scenarios, without requiring that we rewrite all of our software: Ideally, one can get the benefits without switching programming language, applications, operating system, or hardware.

While there is an upper bound on the potential benefits from using an improved middleware, we were able to get significant performance improvements, even though we are only scratching the surface of what is possible: Our middleware only optimizes power management and synchronization, two areas which represent a tiny fraction of the middleware in common use today. Middleware covers an enormous body of software, and we are not the only ones working

on adapting and improving their performance and energy efficiency. Adapting all middleware to their current needs may dramatically improve computational performance, but it will also require a significant development effort from middleware developers. An which will we believe is only feasible through better training in performance evaluation and better tool support for debugging and optimization.

While the gargantuan task of adaptively optimizing all middleware is beyond the scope of this thesis, we can still illustrate how to extend and improve on our work.

6.2 Future directions and limitations

In this thesis we have shown how to predict execution time from CPU frequency with Amdahl's law, and we have shown how to apply and optimize code for hardware transactional memory. We have demonstrated the usefulness of these techniques in our middleware. We can improve the individual middleware components, as described in their respective "Future work" sections, but this section focuses on where we how we further expand on the fundamental techniques. We believe that the prediction model and the optimization guidelines could be applied in far more diverse scenarios.

Our execution time model could be applied to when selecting hardware for a given task, or for task scheduling — ensuring that the least amount of effort is used for each task. By extended the model to incorporate the level of parallelism, we could also schedule tasks efficiently in data centers.

While our optimization guidelines resulted in efficient concurrent code, it is still somewhat uncertain how general they are, and how much the average programmer would benefit from the guidelines. Prior studies have found that the development effort required for transactional memory is lower than the development effort for traditional synchronization. We hope that giving the average programmer some guidelines for how to optimize their code for transactional memory, will further improve their productivity, but so far we have not studied the effects of such guidelines. The guidelines could also be applied automatically, through compiler feedback and optimizations — in fact, Sun Microsystems have patented a compiler optimization pass for moving memory writes to the end of transactions.

This thesis represents one approach for improving computational performance with software, but there are several other reasonable approaches, especially

within compiler optimizations: Static analysis and compiler optimizations have been improving computational performance since the inception of high level languages, and there continues to be several possibilities for improvement. Developments in autoparallelization and high level synthesis — generation of hardware from code in a high level language — have shown that it is possible to greatly improve computational performance and energy efficiency, with little deployment effort. As long as hardware capabilities and software requirements keep evolving, there will be a need to invent new compiler optimizations, or for adapting old optimizations.

Our work has only focused on shared memory systems, although the power governor would likely work as well in a distributed memory setting. Research in computer architecture is increasingly focused on distributed memory systems and highly heterogeneous systems, which can improve computational energy and time efficiency. The computational performance and increasing prevanlence of GPUs and FPGAs make it more attractive to run code on what was once considered accelerator devices. Current computers, especially smart phones, have many of these accelerators, and future computers will likely have even more accelerators, because they represent the most energy efficient way of using an ever increasing number of transistors.

Software, including middleware, will have to adapt to the changing hardware capabilities and consumer requirements — possibly assisted by new programming models, or old programming models adapted for new realities. Software development never really ends, because there will always be new requirements and challenges. Our best bet is to stay alert, stay critical, and stay up to date; measuring, reasoning about, and changing our software and our approaches to keep up with the current needs.

Appendix A

Proof of ELB-tree semantics

This appendix proves correctness of the ELB-trees operations' semantics and that the operations are lock-free.

The following is a brief summary of the design of the data structure, which is detailed in section 3 of the paper. All ELB-trees have a permanent root node r with a single child. ELB-trees are k-ary leaf-oriented search tree, or multiway search trees, so internal nodes have up to k children and $k-1$ keys. An ELB-trees contain a set E_r of integer keys in the range $(0; 2^{63})$. The key 0 is reserved. Keys have an additional read-only bit: when the read-only bit is set, the key cannot be written to. ELB-trees offer 3 main operations:

- Search(e_1, e_2) returns a key e from E_r satisfying $e_1 \leq e \leq e_2$, if such a key exists. Otherwise it returns 0.
- Remove(e_1, e_2) removes and returns a key e from E_r satisfying $e_1 \leq e \leq e_2$, if such a key exists. Otherwise it returns 0.
- Insert(e) adds e to E_r, if e was not in E_r before. If e was in E_r before the behavior is undefined.

ELB-trees can also be used as dictionaries or priority queues by storing values in the least significant bits of the keys.

The operations of ELB-trees cannot generally be expressed as atomic operations, as they occur over a time interval. As a consequence, series of concurrent operations cannot generally be expressed as ocurring serially, that is the semantics are not linearizable. However, the set E_r is atomic. E_r is the union of the keys in the leaf nodes of the ELB-tree. The keys in internal nodes guide tree search.

Section 2 provides formal definitions for terms used throughout the proof. The proof starts in Section 3 by proving that ELB-trees are leaf-oriented search trees. We prove through induction, that ELB-trees are leaf-oriented search trees initially, and that all operations maintain that property. The inductive step is assisted by two significant subproofs:

1. Rebalancing does not change the keys in E_r.
2. The keys in leaf nodes are within a permanent range.

These properties hold due to the behavior of rebalancing. The first subproof shows that rebalancing is deterministic, even when concurrent. The second shows that leaf nodes have a range of keys they may contain and it never changes.

Given these properties, Section 4 derives the operations' semantics. Section 5 follows up by proving that the operations are lock-free. First we prove that some operation has made progress whenever a node is rebalanced. Next we prove that some operation has made progress whenever any part of an operation is restarted.

Section 6 concludes the technical report with a summary.

A.1 Definitions

This section introduces definitions used in the following proofs of the ELB-trees' properties. The definitions start with the terms used, before moving on to the contents and properties of nodes. Finally the intitial state of ELB-trees is formally defined.

Let L be the set of leaf ndoes, I the set of internal nodes, and T the set of points in time. The sets are disjoint.

Nodes contain:

$C_i(t)$ list of children of internal node i at time t

$S_i(t)$ list of keys in internal node i at time t

$E_n(t)$ keys represented by the node n where at time t:

$$E_n(t) = \begin{cases} \text{Non-zero keys in } l & n \in L \\ \bigcup_{c \in C_i(t)} E_c(t) & : n \in I \end{cases}$$

The following node properties can be derived from their content:

$D_n(t)$ the descendants of node n at time t:
$D_n(t) = \begin{cases} \emptyset & : n \in L \\ C_n(t) \cup \bigcup_{d \in C_n(t)} D_d(t) & : n \in I \end{cases}$
n is reachable when $reachable_n(t) \equiv n \in (\{r\} \cup D_r(t))$

$parent_n(t)$ the parents of node n:
$parent_n(t) = \{i \in reachable_r(t) | n \in C_i(t)\}, t \in T$

Initially r has one child $C_r(0) = \langle ic \rangle$, and one grandchild $C_{ic}(0) = \langle ln \rangle$. The grandchild is an empty leaf node $E_{ln}(0) = \emptyset \wedge E_r(0) = \emptyset$.

A.2 Search tree proof

This section proves that ELB-trees are k-ary leaf-oriented search trees. In such a tree, all nodes except the root have one parent, and all internal nodes have strictly ordered keys. Specifically the i'th key in a node provides an upper bound for the i'th child of the node, and a lower bound for the $i+1$'th child. The key ordering is formally expressed as:

$$W_i(t) \equiv \forall j \in [0; C_i(t)). E_{C_i(t)_t} \subseteq (0; S_{ij}] \wedge E_{C_i(t)_t} \subseteq (S_{ij}; 2^{63})$$

The tree property is formally expressed as:

$$\forall n \in reachable_n(t). |parent_n(t)| = 1 \vee n = r$$

The properties are proven inductively, but doing so requires several intermediate steps. To begin with, we will show that the behavior of rebalancing of search trees is deterministic, and does not change E_r.

Lemma A.1 *Unbalanced nodes and their parent are read-only while rebalancing.*

Proof. While finding the nodes involved in rebalancing, they are made read-only: internal nodes are made read-only by setting their status field, and leaf nodes are made read-only by setting the read-only bit of all their keys, see Figure 16 in the paper.

Lemma A.2 *If W_r holds and the unbalanced nodes' parent is still reachable, all threads can find the nodes involved in a rebalancing from the status field of the unbalanced nodes grandparent, .*

Proof. The status field stores the key of the unbalanced node and its parent. Since W_r holds, the nodes can be found by searching for the key in the grandparent and parent of the unbalanced node.

Lemma A.3 *Rebalancing completes deterministically exactly once, if W_r holds.*

Proof. Rebalancing finds the involved nodes (Lemma A.2) and decides how to rebalance (Lemma A.1) deterministically. The parent is replaced, and the grandparent's status field is cleared using ABA safe CAS operations, see Section 3b of the paper. The grandparent has the status field {*,*,*,STEP2} when replacing the parent, ensuring that the grandparent is reachable when replacing the parent node.

Lemma A.4 *$E_r(t)$ does not change when rebalancing, if W_r holds.*

Proof. The content of balanced nodes and their new parent is copied from the old nodes, while their content is read-only (Lemma A.1).

The preceding lemmas show that rebalancing is well-behaved in search trees. The following lemmas will show that all operations maintain the tree property and W_r.

Lemma A.5 *All operations maintain the tree property, if W_r holds.*

PROOF. $descendants_n$ only changes when rebalancing. Specifically, $descendants_n$ changes when replacing an internal node op with a new node np. The children of op had op as their only parent, so all the children np and op share, will have np as their only parent after rebalancing. The new children have np as their only parent, because they have just been introduced, and the descendants of the new nodes have their parents replaced. Formally:

$$(\forall c \in C_{op}(t_1).parent_c(t_1) = \{op\}) \Rightarrow \forall c \in C_{np}(t_2).parent_c(t_2) = \{np\}$$

LEMMA A.6 *Leaf nodes l have a permanent range R_l of keys they may contain, if W_r holds.*

PROOF. The lower bound is given by the keys of its ancestors. The ancestors change deterministically when W_r holds (Lemma A.3). Although the ancestors may change, their replacements use the same keys. Internal node keys are only introduced or removed when splitting and merging nodes, which results in two or three new nodes. When rebalancing results in two new nodes, the new parent has one less key. When rebalancing results in three new nodes, the new parent has one updated or additional key, which the old parent did not have. The updated or new key is copied from its the unbalanced nodes, so it only affects the new nodes.

LEMMA A.7 *If W_r holds, the leaf node l reached by $Search(e, e)$ satisfies: $W_r \Rightarrow e \in R_l$.*

PROOF. Search visiting a node n where $\neg reachable_n(t)$ eventually restarts, so a terminating search only visits reachable nodes in the tree (Lemma A.5). Search of reachable nodes when W_r holds is regular k-ary tree search.

LEMMA A.8 *If W_r holds, searching the leaf node l from t_{l1} to t_{l2} must read the keys $O(t_{l1}, t_{l2}) \cap R_l$.*

PROOF. l is read after a memory barrier, ensuring that $O(t_{l1}, t_{l2}) \cap R_l$ are read.

LEMMA A.9 *All writes to the tree maintain W_r. Formally:*

$$\forall t_1, t_2 \in T.(t_1 \leq t_2 \wedge W_r(t_1)) \Rightarrow W_r(t_2)$$

PROOF. Writes to the tree can be classified into: key insertion, key removal, and rebalancing. Rebalancing maintains W_r (Lemma A.6). Key removal and

insertion only affects the keys in the tree. $remove(e_1, e_2, t_1, t_2)$ removes an key from a leaf node l, which maintain W_r. $insert(e, t_1, t_2)$ inserts into leaf nodes for which $\forall t \in T.W_r(t) \Rightarrow e \in R_l$ (Lemma A.7), which maintain W_r.

Theorem A.10 *ELB-trees are leaf-oriented search trees.*

Proof. ELB-trees are trees and W_r holds initially. All operation on ELB-trees maintains the tree property (Lemma A.5) and W_r (Lemma A.9).

This section proves that ELB-trees are leaf-oriented search trees. Such proofs are sufficient to derive the semantics of concurrent searches and serial insertions and removals. The next section will derive the semantics of the concurrent operations, which requires a few additional lemmas.

A.3 Correctness

This section derives the semantics of the operations. But first we will introduce some terms to reason about the results of such operations. Let:

$search(e_1, e_2, t_1, t_2)$ be the result of a search operation matching against keys $e \in [e_1; e_2]$ starting at t_1 and ending at t_2;

$remove(e_1, e_2, t_1, t_2)$ be the result of a remove operation matching against keys $e \in [e_1; e_2]$ starting at t_1 and ending at t_2;

$insert(e, t_1, t_2)$ be an insert e operation starting at t_1 and ending at t_2;

$O(t_1, t_2)$ be the keys that were in E_r at all times during $[t_1; t_2)$:

$$O(t_1, t_2) = \{e | \forall t \in [t_1; t_2).e \in E_r(t)\}; \text{ and}$$

$U(t_1, t_2)$ be the keys that were in E_r at some time during $[t_1; t_2)$:

$$U(t_1, t_2) = \{e | \exists t \in [t_1; t_2).e \in E_r(t)\}.$$

We first prove properties of search operations, then derive the operations' semantics:

Lemma A.11 *Searching a set of leaf nodes RL from t_1 to t_2 reads the keys $\bigcup_{l \in RL} R_l \cap O(t_1, t_2)$.*

PROOF. The search reads the keys $\bigcup_{l \in RL} R_l \cap O(t_{l1}, t_{l2})$ (Lemma A.8). $\forall l \in RL.O(t_{l1}, t_{l2}) \subseteq O(t_1, t_2)$ holds, as any key in the tree during t_1 to t_2 must have been in the tree for all fragments of that duration.

THEOREM A.12 *$search(e_1, e_2, t_1, t_2)$ can only return 0 (fail) if there are no matching entries in E_r at all times during $[t_1, t_2)$:*

$$search(e_1, e_2, t_1, t_2) = 0 \Rightarrow [e_1; e_2] \cap O(t_1, t_2) = \emptyset$$

PROOF. $search(e_1, e_2, t_1, t_2) = 0$ implies that a set of leaf nodes RL have been searched, where $[e_1; e_2] \subseteq \bigcup_{l \in RL} R_l$. If there was an key in $[e_1; e_2] \cap O(t_1, t_2)$ it would have been read (Theorem A.10, Lemma A.11).

THEOREM A.13 *Successful searches return a matching key that was in E_r at some point in time during $[t_1; t_2)$:*

$$e = search(e_1, e_2, t_1, t_2) \Rightarrow (e \in U(t_1, t_2) \wedge e \in [e_1; e_2])$$

PROOF. Successful searches return a key e that was read from a leaf. Since e was read it must have been in E_r (Lemma A.11).

THEOREM A.14 *Remove can only return 0 (fail) if there are no matching entries in E_r at all times during $[t_1, t_2)$:*

$$remove(e_1, e_2, t_1, t_2) = 0 \Rightarrow O(t_1, t_2) \cap [e_1; e_2] = \emptyset.$$

PROOF. Terminating remove operations that return 0 have searched a set of leafs RL satisfying $[e_1; e_2] \subseteq \bigcup_{l \in RL} R_l$ (Lemma A.11), so any keys in $O(t_1, t_2) \cup [e_1; e_2]$ would have been read.

THEOREM A.15 *Successful remove operations remove matching a key e from E_r that was in E_r at some point in time during $[t_1; t_2)$:*

$$e = remove(e_1, e_2, t_1, t_2) \neq 0 \Rightarrow$$
$$(e_1 \leq e \leq min(O(t_1, t_2) \cap [e_1; e_2]) \leq e_2 \wedge e \in U(t_1, t_2))$$

PROOF. Terminating remove operations have searched a set of leafs RL satisfying $[e_1; e] \subseteq \bigcup_{l \in RL} R_l$ (Lemma A.11). Any keys smaller than e in $O(t_1, t_2) \cup [e_1; e_2]$ would have been read.

Theorem A.16 *$insert(e, t_1, t_2)$ adds e to the E_r, if $e \notin U(t_1, t_2)$.*

Proof. Insert operations terminate when they use a successful CAS operation to write the key into an empty key of a leaf node l where $e \in R_l$ (Lemma A.7). The CAS operations success implies the key is not read-only, and hence $reachable_l(t_2)$.

Theorem 2-6 can be summarized as:

$$e = search(e_1, e_2, t_1, t_2) \Rightarrow \begin{cases} O(t_1, t_2) \cap [e_1; e_2] = \emptyset & : e = 0 \\ e_1 \leq e \leq e_2 \wedge e \in U(t_1, t_2) & : e \neq 0 \end{cases}$$

$$e = remove(e_1, e_2, t_1, t_2) \Rightarrow \begin{cases} O(t_1, t_2) \cap [e_1; e_2] = \emptyset & : e = 0 \\ e_1 \leq e \leq min([e_1; e_2] \cap O(t_1, t_2)) & \\ \wedge e \in U(t_1, t_2) & : e \neq 0 \end{cases}$$

$insert(e, t_1, t_2)$ adds e to E_r, if $e \notin U(t_1, t_2)$.

A.4 Lock-freedom

Lock-freedom guarantees that as long as some thread is working on an operation o_1, some operation o_2 is coming closer to terminating. In this case we say o_1 is causing progress, and o_2 is making progress. The operations o_1 and o_2 can be different. For ELB-trees, this means that whenever a thread is searching, inserting, or removing, some thread must be making progress. The following is proof that the operations are lock-free:

Lemma A.17 *Operations eventually terminate or restart part of their operation.*

Proof. The operations' algorithms have loops in for: node search, tree search, rebalancing, and updating keys in leafs. The algorithms are given in the paper. Without concurrency, they iterate up to k, h, h, and 1 times, where k is the capacity of leafs, and h is the height of the tree. With concurrency, tree search, rebalancing, and key update loops may restart part of their operation.

Lemma A.18 *Rebalancing leaf nodes cause progress.*

Proof. Threads start rebalancing when visiting nodes with sizes $size \leq S \vee D \leq size$. If the nodes are written to between deciding to rebalance and rebalancing, some operation has made progress. If there are no writes, the size of the first node is either D or S, resulting in balanced nodes of $size \in [min(2S, 0.5D); D -$

1]. Such nodes can be removed from and inserted into at least once before requiring additional rebalancing. As such, every time a rebalancing completes, one operation has made progress.

Lemma A.19 *Rebalancing internal nodes cause progress.*

Proof. Rebalancing internal nodes leads to child nodes that can be rebalanced at least one. Each leaf rebalancing cause progress (Lemma A.18), hence each internal rebalancing cause progress.

Theorem A.20 *Search causes progress.*

Proof. Search eventually terminates, similar to k-ary tree search, or restarts part of their operation (Lemma A.7, Lemma A.17). In the first case the search operation is making progress. In the second case some operation is making progress (Lemma A.18, Lemma A.19).

Theorem A.21 *Remove and insert operations cause progress.*

Proof. The operations proceed as searches, which are lock-free (Theorem A.20), followed by writes to leaf nodes. The leaf node write eventually terminates or restarts (LemmaA.17). The write may restart part of the operation either due to rebalancing, or other insertions and removals terminating. In the first case, some operation is nearing termination, and in the second case some operation terminated (Lemma A.18, Lemma A.19).

A.5 Conclusion

This technical report has introduced, proved, and derived properties of ELB-trees. ELB-trees have been proven to be leaf-oriented search trees. Their operations' semantics have been derived as:

$$e = search(e_1, e_2, t_1, t_2) \Rightarrow \begin{cases} O(t_1, t_2) \cap [e_1; e_2] = \emptyset & : e = 0 \\ e_1 \leq e \leq e_2 \wedge e \in U(t_1, t_2) & : e \neq 0 \end{cases}$$

$$e = remove(e_1, e_2, t_1, t_2) \Rightarrow \begin{cases} O(t_1, t_2) \cap [e_1; e_2] = \emptyset & : e = 0 \\ e_1 \leq e \leq min([e_1; e_2] \cap O(t_1, t_2)) & \\ \wedge e \in U(t_1, t_2) & : e \neq 0 \end{cases}$$

$insert(e, t_1, t_2)$ adds e to E_r, if $e \notin U(t_1, t_2)$. Finally the operations have been proven to be lock-free.

Appendix B

TSX Support in Broadwell Processors

This chapter shows which Intel Broadwell processors support TSX as of Juli 17th 2015. In particular, we show how frequently the different Intel brands (Xeon, i5, Celeron, etc.) support TSX, and list the processor models which do not support TSX.

B.1 Desktop Broadwell TSX coverage by branding

Branding	Coverage
i5	3 / 3
i7	2 / 2

B.2 Server Broadwell TSX coverage by branding

Branding	Coverage
Xeon D	2 / 2
Xeon E3	3 / 3

B.3 Mobile Broadwell TSX coverage by branding

Branding	Coverage
Core M	2 / 7
Celeron	0 / 4
Pentium	0 / 2
i3	0 / 5
i5	2 / 7
i7	2 / 9

B.3.1 Mobile Broadwell without TSX

Model	Branding
5y10	Core M
5y10a	Core M
5y10c	Core M
5y31	Core M
5y51	Core M
3205U	Celeron
3215U	Celeron
3755U	Celeron
3765U	Celeron
3805U	Pentium
3825U	Pentium
5005U	i3
5010U	i3
5015U	i3
5020U	i3
5157U	i3
5200U	i5
5250U	i5
5257U	i5
5287U	i5
5350H	i5
5500U	i7
5550U	i7
5557U	i7
5700HQ	i7
5750HQ	i7
5850HQ	i7
5950HQ	i7

B.3.2 Mobile Broadwell with TSX

Model	Branding
5y70	Core M
5y71	Core M
5300U	i5
5350U	i5
5600U	i7
5650U	i7

B.4 Embedded Broadwell TSX coverage by branding

Branding	Coverage
Celeron	0 / 1
i3	0 / 1
i5	1 / 1
i7	3 / 3
Xeon	2 / 2

B.4.1 Embedded Broadwell without TSX

Model	Branding
3765U	Celeron
5010U	i3

B.4.2 Embedded Broadwell with TSX

Model	Branding
5350U	i5
3650U	i7
5700EQ	i7
5850EQ	i7
1258L v4	Xeon E3
1278L v4	Xeon E3

Bibliography

[7-c] 7-cpu. Intel Haswell. [Online; Last accessed http://www.7-cpu.com/cpu/Haswell.html 21/09/2015].

[ALM14] Yehuda Afek, Amir Levy, and Adam Morrison. Software-improved hardware lock elision. In *Symposium on Principles of Distributed Computing (PODC)*, pages 212–221. ACM, 2014.

[Alp15] Seeking Alpha. Intel's CEO Brian Krzanich on q2 2015 results - earnings call transcript, 2015. [Online; http://seekingalpha.com/article/3329035-intels-intc-ceo-brian-krzanich-on-q2-2015-results-earnings-call-transcript?page=2 Last accessed 13/09/2015].

[And90] Thomas E. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990.

[Bag00] Phil Bagwell. Ideal hash trees. Technical report, Computer Science Department, Ecole Polytechnique Federale de Lausanne, 2000.

[Bay72] Rudolf Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta informatica*, 1(4):290–306, 1972.

[BBB+91] David H. Bailey, Eric Barszcz, John T. Barton, David S. Browning, Russell L. Carter, Leonardo Dagum, Rod A. Fatoohi, Paul O. Frederickson, Thomas A. Lasinski, Rob S. Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[BCKT14] Hansang Bae, James Cownie, Michael Klemm, and Christian Terboven. A user-guided locking api for the openmp* application program interface. In *International Workshop on OpenMP (IWOMP)*, pages 173–186. Springer, 2014.

[BDG+04] Jairo Balart, Alejandro Duran, Marc Gonzàlez, Xavier Martorell, Eduard Ayguadé, and Jesús Labarta. Nanos mercurium: a research compiler for openmp. In *Proceedings of the European Workshop on OpenMP*, volume 8, 2004.

[BER14] Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 329–342. ACM, 2014.

[BH11] Trevor Brown and Joanna Helga. Non-blocking k-ary search trees. In *Principles of Distributed Systems*, pages 207–221. Springer, 2011.

[Bie11] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[BKP13] Lars F. Bonnichsen, Sven Karlsson, and Christian W. Probst. ELB-trees an efficient and lock-free B-tree derivative. In *International Workshop on Multi-Many-core Computing Systems (MuCoCoS)*, pages 1–10. IEEE, Sept 2013.

[BM70] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. In *Workshop Data Description and Access*, pages 107–141. ACM, 1970.

[BMSMMA12] Siti Sarah Binti Md. Sallah, Habibah Mohamed, Md. Mamun, and Md. Syedul Amin. CMOS downsizing: present, past and future. *Journal of Applied Sciences Research*, 8(8):4138–4146, 2012.

[BMT+07] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. The OpenTM transactional application programming interface. In *Parallel Architecture and Compilation Techniques (PACT)*, pages 376–387. IEEE, 2007.

[BMV+07] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *International Symposium on Computer Architecture (ISCA)*, pages 81–91. ACM, 2007.

[Boh07] Mark Bohr. A 30 year retrospective on Dennard's MOSFET scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, 2007.

[Bon12] Lars Frydendal Bonnichsen. Contention resistant non-blocking priority queues. Master's thesis, The Technical University of Denmark, 2012.

[BP12] Anastasia Braginsky and Erez Petrank. A lock-free B+tree. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 58–67. ACM, 2012.

[BP15] Lars F. Bonnichsen and Artur Podobas. Using transactional memory to avoid blocking in openmp synchronization directives. In *International Workshop on OpenMP (IWOMP)*. Springer, 2015.

[BPK15] Lars F. Bonnichsen, Christian W. Probst, and Sven Karlsson. Hardware transactional memory optimization guidelines, applied to ordered maps. In *International Symposium on Parallel and Distributed Processing with Applications (ISPA)*. IEEE, August 2015.

[Bro] Trevor Brown. Personal homepage. [Online; http://www.cs.toronto.edu/~tabrown Last accessed 12/20/2014].

[CGR12] Tyler Crain, Vincent Gramoli, and Michel Raynal. A speculation-friendly binary search tree. *Acm Sigplan Notices*, 47(8):161–170, 2012.

[CMF+13] Harold W. Cain, Maged M. Michael, Brad Frey, Cathy May, Derek Williams, and Hung Le. Robust architectural support for transactional memory in the power architecture. In *International Symposium on Computer Architecture (ISCA)*, pages 225–236. ACM, 2013.

[DFGG11] Aleksandar Dragojević, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Communications of the ACM*, 54(4):70–77, 2011.

[DLB59] René De La Briandais. File searching using variable length keys. In *Papers presented at the the March 3-5, 1959, western joint computer conference*, pages 295–298. ACM, 1959.

[DLM+09] Dave Dice, Yossi Lev, Mark Moir, Dan Nussbaum, and Marek Olszewski. Early experience with a commercial hardware transactional memory implementation. Technical report, Sun Microsystems, Inc., 2009.

[DTF+09] Alejandro Duran, Xavier Teruel, Roger Ferrer, Xavier Martorell, and Eduard Ayguade. Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *International Conference on Parallel Processing (ICPP)*, pages 124–131. IEEE, 2009.

[DVY14] Dana Drachsler, Martin T. Vechev, and Eran Yahav. Practical concurrent binary search trees via logical ordering. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 343–356. ACM, 2014.

[ECLV12] Maja Etinski, Julita Corbalán, Jesús Labarta, and Mateo Valero. Understanding the future of energy-performance trade-off via DVFS in HPC environments. *Journal of Parallel and Distributed Computing*, 72(4):579–590, 2012.

[EFRvB10] Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Symposium on Principles of Distributed Computing (PODC)*, pages 131–140. ACM, 2010.

[Fog14] Agner Fog. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs, December 2014. [Online; http://www.agner.org/optimize/instruction_tables.pdf Last accessed 16/09/2015].

[FR75] Robert W. Floyd and Ronald L. Rivest. Algorithm 489: the algorithm select–for finding the i'th smallest of n elements [m1]. *Communications of the ACM*, 18(3):173, 1975.

[FR04] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Symposium on Principles of Distributed Computing (PODC)*, pages 50–59. ACM, 2004.

[Fra04] Keir Fraser. *Practical lock-freedom.* PhD thesis, University of Cambridge, 2004.

[Fre60] Edward Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

[GBCH01] Stephen Gunther, Frank Binns, Douglas M. Carmean, and Jonathan C. Hall. Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal*, 5(1):1–9, 2001.

[GMG+10] Bhavishya Goel, Sally McKee, Roberto Gioiosa, Karan Sing, Major Bhadauria, Marco Cesati, et al. Portable, scalable, per-core power estimation for intelligent resource management. In *International Green Computing Conference and Workshops (IGCC)*, pages 135–146. IEEE, 2010.

[Hen06] John L. Henning. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[HH08] Nor Zaidi Haron and Said Hamdioui. Why is CMOS scaling coming to an end? In *International Design and Test Workshop (IDT)*, pages 98–103. IEEE, 2008.

[HM93] Maurice Herlihy and J. Eliot B. Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[Hoa61] Charles Anthony Richard Hoare. Algorithm 65: find. *Communications of the ACM*, 4(7):321–322, 1961.

[HSN04] Domenik Helms, Eike Schmidt, and Wolfgang Nebel. Leakage in CMOS circuits–an introduction. In *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, pages 17–35. Springer, 2004.

[Int14a] Intel. Intel Xeon processor E3-1200 v3 product family specification update, August 2014. [Online; http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf Last accessed 09/08/2014].

[Int14b] Intel. *Programming with Intel Transactional Synchronization Extensions*, June 2014.

[Int14c] Intel. Thread Building Blocks, 2014. [Online; https://www.threadingbuildingblocks.org/ Last accessed 09/11/2014].

[Int15a] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B: System Programming Guide, Part 2*, June 2015.

[Int15b] Intel. Products (formerly Broadwell), 2015. [Online; http://ark.intel.com/products/codename/38530/Broadwell Last accessed 17/07/2015].

[KBSW11] Jonathan G. Koomey, Stephen Berard, Marla Sanchez, and Henry Wong. Implications of historical trends in the electrical efficiency of computing. *IEEE Annals of the History of Computing*, 33(3):46–54, 2011.

[KEYM14] Shin-gyu Kim, Hyeonsang Eom, Heon Y. Yeom, and Sang Lyul Min. Energy-centric DVFS controlling method for multi-core platforms. *Computing*, 96(12):1163–1177, 2014.

[KK14] Melanie Kambadur and Martha A. Kim. An experimental survey of energy management across the stack. In *International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 329–344. ACM, 2014.

[Koo08] Jonathan G Koomey. Worldwide electricity used in data centers. *Environmental Research Letters*, 3(3):034008, 2008.

[KSK10] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. Interval-based models for run-time DVFS orchestration in superscalar processors. In *International conference on Computing frontiers (CF)*, pages 287–296. ACM, 2010.

[Lev13] Amir Levy. Programming with hardware lock elision. Master's thesis, Tel-Aviv University, September 2013.

[Lit84] Witold Litwin. Trie hashing. In *SIGMOD Conference*, pages 19–29. ACM, 1984.

[LSH10] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *International conference on Power aware computing and systems*, pages 1–8. USENIX Association, 2010.

[LSH11] Etienne Le Sueur and Gernot Heiser. Slow down or sleep, that is the question. In *USENIX Annual Technical Conference*, 2011.

[LSV89] Witold Litwin, Yehoshua Sagiv, and K Vidyasankar. Concurrency and trie hashing. *Acta informatica*, 26(7):597–614, 1989.

[LVHV+12] Sofie Lambert, Ward Van Heddeghem, Willem Vereecken, Bart Lannoo, Didier Colle, and Mario Pickavet. Worldwide electricity consumption of communication networks. *Optics express*, 20(26):B513–B524, 2012.

[M+11] Chris Mack et al. Fifty years of moore's law. *IEEE Transactions on Semiconductor Manufacturing*, 24(2):202–207, 2011.

[Mar99] George Marsaglia. Random numbers for c: The end. *Posted to the electronic billboard sci. crypt. random-numbers*, 1999.

[MBH05] Tali Moreshet, R Iris Bahar, and Maurice Herlihy. Energy reduction in multiprocessor systems using transactional memory. In

Low Power Electronics and Design, 2005. ISLPED'05. Proceedings of the 2005 International Symposium on, pages 331–334. IEEE, 2005.

[MFG+08] Miloš Milovanović, Roger Ferrer, Vladimir Gajinov, Osman S Unsal, Adrian Cristal, Eduard Ayguadé, and Mateo Valero. Nebelung: execution environment for transactional openmp. *International Journal of Parallel Programming*, 36(3):326–346, 2008.

[MFU+08] Miloš Milovanović, Roger Ferrer, Osman S Unsal, Adrian Cristal, Xavier Martorell, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Transactional memory and openmp. In *A Practical Programming Model for the Multi-Core Era*, pages 37–53. Springer, 2008.

[Mic02] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 73–82. ACM, 2002.

[Mic04] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):491–504, 2004.

[Mil13] Mark P. Mills. The cloud begins with coal. *Digital Power Group. Online at: http://www. tech-pundit. com/wp-content/uploads/2013/07/Cloud_ Begins_ With_ Coal. pdf*, 2013.

[MN98] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[MNP02] Alain J Martin, Mika Nyström, and Paul I Pénzes. Et2: A metric for time and energy efficiency of computation. In *Power aware computing*, pages 293–315. Springer, 2002.

[MR85] Yehudit Mond and Yoav Raz. Concurrency control in B+-trees databases using preparatory operations. In *International Conference on Very Large Data Bases (VLDB)*, pages 331–334, 1985.

[MT02] José F. Martínez and Josep Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *ACM SIGOPS Operating Systems Review*, volume 36, pages 18–29. ACM, 2002.

[MVHS10] Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. Contention aware execution: online contention detection and response. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 257–265. ACM, 2010.

[NM14] Aravind Natarajan and Neeraj Mittal. Fast concurrent lock-free binary search trees. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 317–328. ACM, 2014.

[NSS91] Otto Nurmi and Eljas Soisalon-Soininen. Uncoupling updating and rebalancing in chromatic binary search trees. In *Symposium on Principles of Database Systems (PODS)*, pages 192–198. ACM, 1991.

[Ora14] Oracle. ConcurrentHashMap, 2014. [Online; http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.htm Last accessed 09/11/2014].

[PAT11] Victor Pankratius and Ali-Reza Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 43–52. ACM, 2011.

[PBBO12] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 151–160. ACM, 2012.

[PBV14] Artur Podobas, Mats Brorsson, and Vladimir Vlassov. Turbobłysk: Scheduling for improved data-driven task performance with fast dependency resolution. In *International Workshop on OpenMP (IWOMP)*, pages 45–57. Springer, 2014.

[PK11] Stavros Passas and Sven Karlsson. Comparing the overhead of lock-based and lock-free implementations of priority queues. In *Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, page 78, 2011.

[PRV11] Hari K Pyla, Calvin Ribbens, and Srinidhi Varadarajan. Exploiting coarse-grain speculative parallelism. In *ACM SIGPLAN Notices*, volume 46, pages 555–574. ACM, 2011.

[Pug90] William Pugh. Concurrent maintenance of skip lists. Technical Report TR-CS-2222, Dept. of Computer Science, University of Maryland, 1990.

[RHW10] Christopher J Rossbach, Owen S Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 47–56. ACM, 2010.

[SATH+06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197. ACM, 2006.

[Sed08] Robert Sedgewick. Left-leaning red-black trees. In *Dagstuhl Workshop on Data Structures*, page 17, 2008.

[SEH11] Andreas Sembrant, David Eklov, and Erik Hagersten. Efficient software-based online phase classification. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 104–115. IEEE, 2011.

[SHM12] Robert Schöne, Daniel Hackenberg, and Daniel Molka. Memory performance at reduced cpu clock speeds: an analysis of current x86 64 processors. In *International conference on Power-Aware Computing and Systems (HotPower)*. USENIX Association, 2012.

[SKK11] Vasileios Spiliopoulos, Stefanos Kaxiras, and Georgios Keramidas. Green governors: A framework for continuously adaptive DVFS. In *International Green Computing Conference and Workshops (IGCC)*, pages 1–8. IEEE, 2011.

[SLSPH09] David C Snowdon, Etienne Le Sueur, Stefan M Petters, and Gernot Heiser. Koala: A platform for os-level power management. In *European conference on Computer systems (EuroSys)*, pages 289–302. ACM, 2009.

[SON00] Takayuki Sato, Kazuhiko Ohno, and Hiroshi Nakashima. A mechanism for speculative memory accesses following synchronizing operations. In *Intenational Parallel and Distributed Processing Symposium (IPDPS)*, pages 145–154. IEEE, 2000.

[Sri15] E Srikanth. Zynq-7000 AP SoC low power techniques part 3 - measuring ZC702 power with a standalone application tech tip, June 2015. [Online; http://www.wiki.xilinx.com/Zynq-7000+AP+SoC+Power Last accessed 07/09/2015].

[SS06] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *Journal of the ACM (JACM)*, 53(3):379–405, 2006.

[SSK12] Vasileios Spiliopoulos, Andreas Sembrant, and Stefanos Kaxiras. Power-sleuth: A tool for investigating your program's power behavior. In *International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 241–250. IEEE, 2012.

[ST04] Håkan Sundell and Philippas Tsigas. Scalable and lock-free concurrent dictionaries. In *Symposium on Applied Computing (SAC)*, pages 1438–1445. ACM, 04.

[ST97] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[TDK] Ami Tavory, Vladimir Dreizin, and Benjamin Kosnik. Policy-based data structures. [Online; https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/ Last accessed 15/03/2012].

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. *ACM SIGARCH Computer Architecture News*, 23(2):392–403, May 1995.

[VHLL+14] Ward Van Heddeghem, Sofie Lambert, Bart Lannoo, Didier Colle, Mario Pickavet, and Piet Demeester. Trends in worldwide ICT electricity consumption from 2007 to 2012. *Computer Communications*, 50:64–76, 2014.

[WAG+14] Michael Wong, Eduard Ayguadé, Justin Gottschlich, Victor Luchangco, Bronis R de Supinski, Barna Bihari, et al. Towards transactional memory for openmp. In *International Workshop on OpenMP (IWOMP)*, pages 130–145. Springer, 2014.

[WM+08] Vincent M. Weaver, Sally McKee, et al. Can hardware performance counters be trusted? In *International Symposium on Workload Characterization (IISWC 2008)*, pages 141–150. IEEE, 2008.

[WOT+95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. *ACM SIGARCH Computer Architecture News*, 23(2):24–36, 1995.

[WTM13] Vincent M. Weaver, Dan Terpstra, and Steven Moore. Non-determinism and overcount on modern hardware performance counter implementations. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 215–224. IEEE, 2013.