UNIVERSITY OF CALIFORNIA, SAN DIEGO

# Exploiting Non-Traditional Parallelization for Application Performance and Energy Efficiency in Parallel Systems

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Md Kamruzzaman

Committee in charge:

Professor Dean Michael Tullsen, Chair
Professor Steven Swanson, Co-Chair
Professor Pamela C. Cosman
Professor Ranjit Jhala
Professor Andrew B. Kahng

2013

## DEDICATION

To my dear parents and my beloved wife.

EPIGRAPH

*O you who believe! Stand out firmly for Allah and be just witnesses and let not the enmity and hatred of others make you avoid justice. Be just: that is nearer to piety, and fear Allah. Verily, Allah is Well-Acquainted with what you do.*

— Al-Quran, Surat Al-Mā'idah verse 8

TABLE OF CONTENTS

# LIST OF FIGURES

LIST OF TABLES

## ACKNOWLEDGEMENTS

Alhamdulillah. First, I want to thank the Almighty Allah for giving me the opportunity to fulfill my dream to have a Ph.D. I want to acknowledge all the people who constantly supported me to arrive at this stage. I would like to thank my advisers – Dean Tullsen and Steven Swanson. I am lucky to have such awesome advisers. Their inspiration and guidance helped me to grow up as a researcher. They put faith in me even when things were not going well and always encouraged me. Without their support, this dissertation would not have been possible.

I want to thank the committee members – Pamela C. Cosman, Ranjit Jhala, and Andrew B. Kahng for their feedback and suggestions. It was a great honor to have such esteemed professors in my thesis committee. I also thank Julie Conner for her splendid job as a graduate student adviser.

I would like to thank my mother, Aleya Begum and my father, the late Abdur Rob. They always gave maximum priority to my education despite all the hardship. Their love has been a great impetus for me to go forward. I want to specially acknowledge my mother who single-handedly took the entire burden after the death of my father, but never compromised with my pursuit of education. Without her sacrifice, I would never come that far.

I would like to thank my wife, Sajia Akhter for being my wife and providing me the endless support. She is a perfect companion for me. Despite all the hard work and stress during the research and job hunting, I never lost the spirit because of her. She has been the source of simplicity, serenity, and strength for me.

I would like to thank other family members who gave me courage and supported me at different stages of my life. I want to thank my only sister, Shammi Bari whom I spent a great deal of eventful time in the childhood. Those memories are always priceless for me. I want to thank my little aunt, Farzana Jamal who never asked any question when I wanted something. I want to thank my father-in-law, Dr. Sana Ullah who has been a great man and a great advocate for pursuing the higher education. I also thank my uncles, aunts, cousins, in-laws for their support and inspiration. I feel fortunate to be surrounded by so many good people.

Finally, I would like to thank the folks in the UCSD computer architecture lab – Jeff, Jack, Matt, Leo, Vasileios, Rick, Hung-Wei, and Adrian, friends in BUET – Mahbub, Ashique, Farhan, Nejhum, Sajjad, Shaikat, Sarwar, Irfan, Hasib, Tanvir, Sunny, Ejaj, Nilothpal, Pavel, Ahsan, and Uzzal, friends in Bangladesh, friends in San Diego, and friends all over the world. You all are inspirations to me.

Chapters 1, 5, and 8 contain material from "Coalition Threading: Combining Traditional and Non-Traditional Parallelism to Maximize Scalability", by Md Kamruzzaman, Steven Swanson and Dean M. Tullsen, which appears in PACT'12: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, September 2012. The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2012 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or email permissions@acm.org.

Chapters 3 and 8 contain material from "Software Data Spreading: Leveraging Distributed Caches to Improve Single Thread Performance", by Md Kamruzzaman, Steven Swanson and Dean M. Tullsen, which appears in PLDI'10: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation. The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or email permissions@acm.org.

Chapters 4, and 8 contain material from "Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads", by Md Kamruzzaman, Steven Swanson and Dean M. Tullsen, which appears in ASPLOS'11: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2011 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept., ACM, Inc., fax +1 (212) 869-0481, or email permissions@acm.org.

Chapter 6 contains material from "Underclocked Software Prefetching: More Cores, Less Energy", by Md Kamruzzaman, Steven Swanson and Dean M. Tullsen, which appears in IEEE Micro, July-Aug. 2012, Volume 32, Issue: 4, Page(s): 32-41. The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2012 by the Institute of Electrical and Electronics Engineers (IEEE). Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

## VITA AND PUBLICATIONS

| | |
|---|---|
| 2004 | B. Sc. in Computer Science and Engineering<br>Bangladesh University of Engineering and Technology |
| 2004-2006 | Lecturer<br>American International University-Bangladesh |
| 2006 | Lecturer<br>Bangladesh University of Engineering and Technology |
| 2006-2012 | Graduate Student Researcher<br>University of California, San Diego |
| 2007 | Internship<br>Qualcomm Corporate R&D<br>San Diego, California |
| 2009 | M.S. in Computer Science<br>University of California, San Diego |
| 2010 | C. Phil. in Computer Science<br>University of California, San Diego |
| 2013 | Ph. D. in Computer Science<br>University of California, San Diego |

Md Kamruzzaman, Steven Swanson, Dean M. Tullsen, "Coalition Threading: Combining Traditional and Non-Traditional Parallelism to Maximize Scalability", *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2012.

Md Kamruzzaman, Steven Swanson, Dean M. Tullsen, "Underclocked Software Prefetching: More Cores, Less Energy", *IEEE Micro*, August 2012.

Md Kamruzzaman, Steven Swanson, Dean M. Tullsen, "Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads", *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2011.

Md Kamruzzaman, Steven Swanson, Dean M. Tullsen, "Software Data Spreading: Leveraging Distributed Caches to Improve Single Thread Performance", *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2010.

ABSTRACT OF THE DISSERTATION

# Exploiting Non-Traditional Parallelization for Application Performance and Energy Efficiency in Parallel Systems

by

Md Kamruzzaman

Doctor of Philosophy in Computer Science

University of California, San Diego, 2013

Professor Dean Michael Tullsen, Chair
Professor Steven Swanson, Co-Chair

Multicore processors have become ubiquitous in today's computing platforms, extending from smartphones to data centers. However, exploiting the parallelism that they offer remains difficult, especially for legacy applications and applications with large serial components. Even many parallel applications fail to leverage the ample hardware parallelism and observe scalability limits. This creates a gap between the available hardware and the effective software parallelism. The scenario known as the *parallelization wall* impedes the performance growth that every processor generation used to bring in.

The challenge, then, is to develop techniques that allow multiple cores to work in concert to accelerate a single thread. This dissertation proposes three such techniques – software data spreading, inter-core prefetching, and load-balanced pipeline parallelism – and evaluates them on state of the art real systems. These techniques are software only and exploit application level information to best utilize the underlying hardware.

Software data spreading migrates a thread intelligently to spread the working set over the aggregate space from different private caches. This reduces expensive cache misses and dramatically improves performance along with energy efficiency when the working set fits in the aggregate cache space. Inter-core prefetching uses one or more helper threads to prefetch data in advance and uses thread migrations to access that data locally. This dissertation extends inter-core prefetching further and introduces two more techniques – underclocked software prefetching and coalition threading. The former exploits the decoupled execution model of inter-core prefetching to save power. It applies dynamic frequency scaling on the helper thread to leverage its insensitivity to frequency and allows low frequency helper threads to bring the same performance benefits of high frequency helper threads. The latter technique, coalition threading, explores the potential of applying inter-core prefetching on top of traditional parallelism to improve scalability of parallel applications. Finally, this dissertation discusses load-balanced pipeline parallelism that analytically shows how to exploit loop level pipelining to its maximum potential.

# Chapter 1

# Introduction

Multi-core processors have become ubiquitous over the past decade. The improvements in CMOS process technology governed by Moore's law bring more transistors in each process generation. Until recently, the microprocessor industry built larger superscalars with increased clock frequency to exploit the additional transistors, and this catered to the need for more and more performance improvements for applications. However, this trend is no longer possible because of the increase in power density. Transistor feature size shrinks according to Moore's law, allowing more transistors on a fixed die area, but the operating voltage does not scale down any further due to the leakage and other physical constraints. This triggered the discontinuation of building more powerful superscalars and shifted the direction towards building multicores.

As a result, multicores have gone from being the domain of high-end servers and specialized high-performance computing systems, to becoming ubiquitous in every type of computing platform from smart phones to the data center. Current mainstream offerings contain 4 to 16 execution cores on each processor chip [McG06, neh08], and there is no sign of the trend toward higher core counts slowing. While these architectures have delivered the potential for scalable parallel performance, the impact at the application level has been uneven – software parallelism is not nearly as pervasive as hardware parallelism. There are some environments that provide near-infinite parallelism. Other environments and workloads scale poorly or not at all. If we are to continue to provide performance scaling in this era of increasing hardware parallelism, we must scale application performance both in the presence of abundant software parallelism and when that parallelism is harder to find.

In the multicore era, we can no longer depend on instruction-level parallelism (ILP) and clock frequency as the primary sources of performance improvements. Rather, we need to focus on exploiting thread-level parallelism (TLP). However, several factors make it challenging, including the abundance of non-parallelized legacy code, the difficulty of parallel programming, and the inherent serial nature of many programs and algorithms. Even in applications that are amenable to parallelization, the applicability of multicore processors has its bounds: as manufacturers supply an increasing number of cores, more and more applications will discover their scalability limits due to different types of parallelization complexities including synchronization, cache coherence, and load balancing. Furthermore, Amdahl's Law dictates that the more hardware parallelism is available, the more critical sequential performance becomes. So, cramming more cores on die does not always imply getting more performance. This scenario is known as the *parallelization wall* or *parallelization crisis*. In another way, parallelization wall refers to the lack of parallel threads to enable performance scaling.

The parallelization wall limits the improvements in application performance that we used to have with each processor generation. The problem is likely to deepen and become widespread as multicores are making their way into all sorts of computing platforms, and we move from multicore to many-core architectures. State of the art processors try to address this problem by introducing turbo cores, and specialized on-chip accelerators like GPUs. Turbo cores run in higher than normal frequency when other cores remain idle and can improve sequential performance. Accelerators target special type of codes (e.g., highly parallel kernels) and offload the computation of the processing cores. However, these solutions are effective in limited cases, and are far from being a generic solution to the parallelization wall. The parallelization wall stands as a key bottleneck to the demand of application performance growth.

This dissertation characterizes the parallelization wall problem and introduces several techniques to address the key challenge – how to improve single thread performance on parallel hardware (e.g., multicores). In this dissertation, we mainly describe three different techniques – *software data spreading* (DS), *inter-core prefetching* (ICP), and *load-balanced pipeline parallelism* (LBPP). The first two techniques are examples of *non-traditional parallelization* because they do not distribute the computation over multiple threads (i.e., only one thread computes at a time). All three techniques use more than one core per single thread and improve its performance. This is definitely effective for sequential applications, because there are typically lots of idle cores available for use. The techniques even help when the cores are not otherwise free to use (e.g., parallel applications). There are two main reasons why it is effective. First, sometimes these techniques provide more speedup than what the traditional parallelization gives (i.e., more than 2× improvements using two cores). Second, using fewer computing threads, but accelerating each of them, has a positive impact on the parallelization complexity. So, accelerating single thread execution using multiple cores provide a generic solution against the parallelization wall.

The first technique, data spreading, causes single thread computation to migrate among multiple cores and spread data in a controlled way. This aggregates the private cache spaces of the participating cores and provides the appearance of a bigger private cache for the single thread execution. This improves performance by increasing cache locality and avoiding expensive DRAM or next level cache accesses.

Inter-core prefetching combines thread migration and helper threads to enable a novel technique that allows remote prefetching but local access of data. In inter-core prefetching, helper threads run in separate cores to prefetch data and then use migration to move the main thread (the thread that does the computation) to the prefetched data. The main thread thus migrates to a core with the data it is about to access completely preloaded into the private cache(s), while the helper thread is moved to a new core to begin prefetching the next set of data. Thus, all the cache misses disappear from the critical execution path and appear as local hits. This gives significant speedup to the single thread execution irrespective of the working set size or the data access pattern.

Load-balanced pipeline parallelism exploits the pipeline parallelism available in the loop iteration level. The iteration of a serial loop decomposes into a set of one or more pipeline stages, and some of these stages can be parallel. LBPP provides an execution model that ensures maximum possible parallelism for any number of execution cores. LBPP executes the serial loop in a data parallel fashion, but satisfies all the dependencies using a token based synchronization mechanism. The technique provides linear speedup to a number of serial loops, especially when the parallel stages dominate the sequential stages.

For any performance optimization technique, power/energy consumption is also a concern. All three techniques described above improve energy consumption. Data spreading reduces both power and execution times. Inter-core prefetching and load-balanced pipeline parallelism increase power, but the reduction in execution time compensates that by decreasing the overall consumption of energy. This dissertation also proposes the concept of *underclocked software prefetching* that exploits another window of opportunity in inter-core prefetching to reduce the power consumption. ICP naturally produces heterogeneous threads by decoupling the memory accesses. The computing thread is cpu intensive, but the prefetching thread is memory intensive and less sensitive to frequency. Underclocked software prefetching leverages such heterogeneity by applying frequency scaling on the memory intensive prefetching thread. This saves power without the loss of performance.

Accelerating a single thread using multiple cores solves the problem of having disparity between the abundant hardware parallelism and the lack of effective software parallelism. It also opens up new opportunities for parallel applications. This dissertation introduces *coalition threading* that intelligently combines traditional parallelization techniques with non-traditional parallelization techniques for better scalability. We analyze coalition threading in the context of data parallelism and inter-core prefetching and find that coalition threading works better than data parallelism alone. The key challenge in this case is to decide when to apply non-traditional parallelism because traditional parallelization may outperform them in several cases. We find that using microarchitectural information, we can construct a powerful heuristic that automatically identifies the cases where we should apply inter-core prefetching.

This dissertation demonstrates that software only solutions handle the parallelization wall more effectively when they adapt themselves to different classes of applications. Because, there are more opportunities to engage multiple cores and to exploit locality, and decoupling.

**Software only approaches** Unlike a number of previous approaches that share the same spirit with these, all techniques presented in this dissertation are software only and can be automatically implemented in the compiler. This has several advantages. First, there is no need to build new hardware, and we can use machines that are already available. Second, these techniques apply to a variety of architectures. Third, it gives programmers more flexibility. It is easy to turn off the optimizations when they do not work. In addition, programmers can always understand the techniques and fine-tune the programs. It is also easier to combine several software techniques to extract more benefit.

**Adapting to Applications** The key challenge to accelerate single thread execution using multiple cores is to find ways to engage more than one core effectively. Extracting thread level parallelism and doing computation in parallel is the obvious way. However, for a lot of serial code, this is not possible, and we need alternative ways to exploit the resources of multiple cores.

The techniques presented in this dissertation leverage application characteristics to find ways to employ multiple cores. Data spreading capitalizes on the fact that some applications have working sets that are larger than a single cache, and the applications repeatedly access those data. So, spreading the working set across different cores' caches gets the benefit of cache locality after the first round of access. Inter-core prefetching leverages the fact that it is possible to predict future memory references for a lot of serial computations. So, it decouples the memory access part into a different thread(s), and executes these thread(s) in parallel with the serial computation. Inter-core prefetching also shows sensitivity to other application level information such as ratio of compute and memory operations, data access pattern (sequential, or randomized), percentage of data sharing between two chunks of computations, etc. Availability of this information helps better tuning and makes inter-core prefetching more effective. Finally, load-balanced pipeline parallelism exploits the fact that serial loops may have fine-grain pipeline parallelism and further capitalizes on the fact that some of the pipeline stages are parallel.

**Locality** All three techniques – data spreading, inter-core prefetching, and load-balanced pipeline parallelism improve cache locality. Data spreading and inter-core prefetching ensure the availability of data in a particular core's cache and then bring the computation there using thread migration. This essentially implements *computation follows data* rather than the traditional way of executing *data follows computation.* Load-balanced pipeline parallelism also maintains the communication between stages through the local cache. Locality is important from a performance as well as from a power/energy point of view, because data communication between different parts of the memory subsystem consumes power. Locality reduces pressure on the off-chip bandwidth and ensures better scalability. In addition, locality makes these techniques easily applicable on multiprocessor systems where the communication between different processors is expensive.

**Decoupling** Data spreading extracts no thread level parallelism and uses one core at a time. However, it engages multiple cores by using their caches. Inter-core prefetching and load-balanced pipeline parallelism use multiple cores at a time by decoupling the computation into different parts. Other than extracting thread level parallelism, decoupling sometimes also increases instruction-level and memory-level parallelism. This results in single thread speedup beyond the traditional parallelization limit. However, effective implementation of decoupling requires precise synchronization. ICP and LBPP use *chunking* (clustering similar types of operations) to amortize the synchronization overhead. Chunking is also closely related with locality. By controlling the size of chunks, we can confine the memory footprints within the size of a particular level of cache.

## 1.1 Organization of the dissertation

This dissertation presents techniques that understand the application closely from static and profile guided analysis and exploit the information using a variety of techniques including thread migration, software prefetching, frequency scaling, etc. to use state of the art parallel hardware effectively. Over the course of this dissertation, we first understand the nature of the parallelization wall and then describe the techniques in detail. We organize our dissertation in the following way.

Chapter 2 describes the parallelization wall in detail – the main reasons behind it, the impact on application performance, and the importance of accelerating single thread execution.

Chapter 3 demonstrates the details of software data spreading. We first show the mechanism of the technique and then describe the compiler algorithm that analyzes the data sharing patterns to transform a code automatically to exploit data spreading. We also present the impact on performance and energy for several real systems.

Chapter 4 explains inter-core prefetching in detail. We describe the software infrastructure that supports fast thread migration and allows the effective implementation of inter-core prefetching. We investigate the interactions between chunking granularity, number of helper threads, compute density, data access pattern, and data sharing in the context of inter-core prefetching.

Chapter 5 describes how we can extend inter-core prefetching for an effective implementation of coalition threading. We present the compiler infrastructure to apply coalition threading on pre-parallelized code. We analyze the impact of applying different parallelization techniques on the loop level and demonstrate the compiler heuristic that chooses the right technique.

Chapter 6 discusses underclocked software prefetching. We demonstrate a theoretical model that shows how dynamic frequency scaling can exploit the decoupling and rate mismatch of prefetching and execution created by inter-core prefetching described in Chapter 4. We also analyze the interaction between number of helper threads, size of chunks, different power states, and the latency to change frequencies.

Chapter 7 describes the execution model of load-balanced pipeline parallelism. We develop the theoretical model that compares the parallelism achieved by load-balanced pipeline parallelism and another competing technique, *decoupled software pipelining*. We investigate a thorough evaluation on two state of the art architectures and justify the results predicted by the theoretical model.

Chapter 8 summarizes prior approaches (both hardware and software based) related to our work. Finally, Chapter 9 summarizes the contribution of this dissertation – introduction of the techniques that accelerate single thread execution using multiple cores to handle the parallelization wall problem.

## Acknowledgments

Portions of this research were funded by the US National Science Foundation grants CCF-1018356 and NSF-0643880 and by SRC Grant 2086.001.

This chapter contains material from "Coalition Threading: Combining Traditional and Non-Traditional Parallelism to Maximize Scalability", by Md Kamruzzaman, Steven Swanson and Dean M. Tullsen, which appears in PACT'12: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, September 2012. The dissertation author was the primary investigator and author of this paper.

# Chapter 2

# Parallelization Wall

In this chapter, we define and analyze the parallelization wall that motivates this dissertation. We investigate the primary causes of the wall and the depth of the problem. We explain why accelerating a single thread using multiple cores is the most effective way to address this challenge.

## 2.1 Parallelization Wall

The parallelization wall refers to the phenomenon that application performance does not always scale with the core count. The exponential improvement of application performance that we used to see until very recently has now reached a plateau despite the processor industry adding more cores on a single die each generation.

The parallelization wall has roots in the increasing gap between complex hardware and software abstraction. Processors have evolved in different dimensions since Intel introduced its first 4-bit processor in 1971 – from in-order scalar processor to out-of-order super-scalar and from single processing unit to on-chip multithreading to multicores. Software and programming models, on the other hand, target a very basic hardware abstraction and remain oblivious of the change in hardware. This gives the compiler and runtime system the bulk of the responsibility to extract maximum performance from today's sophisticated hardware.

Three things manifest as the key bottlenecks that impede the application

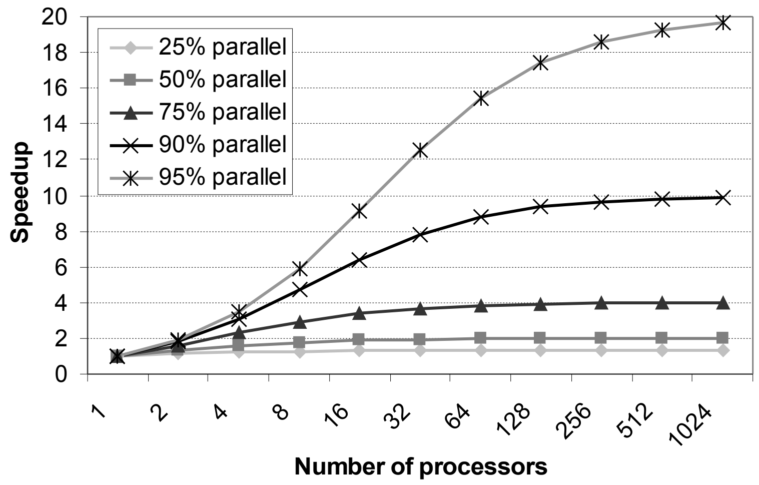


Figure 2.1: Amdahl's law: Serial portion becomes dominant as we increase the number of cores.

performance in parallel hardware and contribute towards the parallelization wall –

- Abundance of serial code
- Parallelization complexity
- Resource constraints

Next, we describe these things in detail.

### 2.1.1 Abundance of serial code

Serial code uses only one processing core and does not see benefit from additional cores. So, putting more cores on die does not normally improve the serial execution as it does for the parallel execution. Amdahl's law explains this circumstance clearly and dictates that as architectures become more parallel, the inherently serial portions of applications will eventually limit the performance. According to Amdahl's law, if $P$ is the parallel portion of a program, and $(1-P)$ is the serial part, then the maximum possible speedup by $N$ processors is –

$$S(N) = \frac{1}{(1-P) + \frac{P}{N}}$$

So, the serial portion provides an upper bound of $\frac{1}{(1-P)}$ for the maximum possible speedup. Figure 2.1 shows the scenarios for different values of $P$ and $N$. In every case, the speedup reaches an asymptote, and the serial part starts taking the bulk of the execution time.

The abundance of serial code is a key component of the parallelization wall. Serial code exists to some extent even in the most embarrassingly parallel applications. There are two reasons why serial code is pervasive. First, some applications or some part of the applications are inherently serial. These codes (e.g., pointer chasing, graph algorithms, recursive code, etc.) cannot be made parallel without changing algorithms or compromising the correctness. Second, plenty of legacy code is serial in nature. These codes might have inherent parallelism, but exposing that requires automatic parallelization in the compiler or manual parallelization. Automatic parallelization is still in its nascent stage and is not always effective. On the other hand, manual parallelization is hard for legacy code as well as for most newly developed applications. As a result, a large amount of serial code is likely to persist.

### 2.1.2 Parallelization complexity

Parallelization is not free and introduces several overheads. Consequently, most parallel codes do not achieve perfect scalability, and if we scale far enough, all parallel codes will experience this. Parallelization overheads result from synchronization, load balancing, cache coherence, and work efficiency.

In most parallel code, threads need to synchronize to maintain data consistency and avoid race conditions. Locks, barriers, thread join, condition variables, etc. are different ways to implement synchronization. The overhead of synchronization depends on the number of participating threads. So, synchronization overhead increases substantially as we scale more and eventually becomes dominant since the total work to be done remains constant.

Load balancing is another challenging task for parallelization. If threads are not load balanced, some threads become the bottleneck and impede the scalability. Also, each thread should at least do a minimum amount of work to amortize the thread spawning overhead. As we scale more, each thread has less work to do, and the relative weight of thread spawning compared to the work to be done increases.

Some parallel computations are not work efficient. In this case, the total amount of work increases as we scale more. OpenMP reduction [CJP07] is an example of that. The reduction part increases with the number of threads. In some cases, the space requirements also increase and put an upper bound on the scalability.

There are hardware overheads as well, especially in the form of cache coherence. In current systems, keeping data coherent is expensive because of costly cache to cache transfers [HMN09]. Coherence activity occurs for both true and false sharing of data and increases when we add more threads. For example, if we use two threads for a simple data parallel loop that writes an array, there can be one falsely shared cache line. However, if we use 64 threads, the number of falsely shared cache lines can be as large as 63.

In summary, with the current parallel execution model there are both significant software and hardware overheads. These overheads are less noticeable when the computation is large and threads can do enough work before doing any synchronization activity. However, in other cases, these overheads tend to dominate as we scale and at some point, start causing negative scalability. The overheads also prohibit us from exploiting small pockets of parallelism. For example, if a critical inner loop is parallel but only iterates for a hundred instructions, we may not leverage the parallelism, because the thread spawning, joining, and false sharing overheads will likely exceed the benefits of parallelism.

### 2.1.3 Resource constraints

Hardware resource constraints in multicores is another cause of the parallelization wall. The processing cores in a multicore share several on-chip structures like last level cache, interconnect, and off-chip bandwidth. These shared structures might become the bottleneck when several cores actively use them, because processor designers normally under-provision these resources for different reasons.

First, there are design constraints (area, power, latency, etc) and physical limitations that prevent the scaling of these resources in the same way we can scale the number of cores. For example, a large cache will incur more latency than a small cache does. Faster interconnect requires more area and power. Increasing off-chip bandwidth is difficult, because there are physical limitations on the number of pins that we can put on-chip [BGK96].

Second, the workloads are diverse. Some are cpu-intensive, and some are memory-intensive. Some applications have strong locality and are very sensitive to the cache architecture, while some applications do not have any locality and are sensitive to the DRAM access latency. There are multithreaded applications with lots of shared data to stress the coherence mechanism. On the other hand, there are also multiprogrammed workloads that do not require cache coherence. The diversity of the workloads make the design of a general purpose multicore challenging. Increasing a shared resource far enough to make it useful for a particular type of applications will not help other applications. As an example, increasing the size of the last level cache may help memory-intensive applications, but will not benefit cpu-intensive applications and will rather consume unnecessary power.

Finally, there is the emerging problem of dark silicon [EBSA$^{+}$11], or utilization wall [VSG$^{+}$10]. For a large multicore, it may not be possible to run all cores at the highest frequency, because of the power constraints. So, we may either run few cores in the highest frequency or run more cores in a lower frequency. State of the art architectures support this form of execution, e.g., AMD turbo cores.

## 2.2 Handling the parallelization wall

The parallelization wall highlights the fact that there is a lack of effective software parallelism to match the ample hardware parallelism. This is obvious for serial code, but the problem also persists for many parallel applications or multiprogrammed workload when executing with a large number of cores. In that case, the software parallelism does not provide the expected performance benefit due to the parallelization complexity and resource constraints described above.

Reducing parallelization complexities like synchronization overhead, cache coherence, etc., will improve the scalability of parallel execution in state of the art hardware. However, complete elimination of these complexities is not possible even for newly developed code. Eliminating resource constraints will also help in limited cases, but as explained earlier, the opportunity cost is much larger.

This leaves us with the final solution – accelerating single thread execution using multiple cores. Unlike the approaches that only improve parallel execution, this improves both serial and parallel execution as explained in Chapter 1. This also gives more design choices to adapt different scenarios. In underclocked software prefetching, we have the options to choose between a small number of fast cores or a large number of slow cores. In coalition threading, we can control the number of cores per single thread execution for better distribution of execution resources. In summary, using more than one core per single thread to improve performance not only solves the parallelization wall problem to a greater extent, but also provides another degree of freedom.

In the following chapters, we will describe and evaluate our techniques in detail.

# Chapter 3

# Software Data Spreading: Leveraging Distributed Caches for Locality

In this chapter, we demonstrate the concept of software data spreading which exploits the capacity of distributed caches to accelerate a single thread. By migrating the thread among multiple cores with distinct caches, we can utilize the combined cache space of all of those cores. Aggregating cache capacity is of growing importance: Although total on-chip cache capacity continues to grow with Moore's law, the *per-core* cache capacity is not keeping pace (e.g., 4MB total for the Intel Core 2 Duo at introduction vs. 8MB total for a quad-core Nehalem chip). Previous work [CS06, M. 05, HKS$^+$05, LSK04] has attempted to aggregate cache space through specialized hardware support.

Migrating a thread among multiple cores while it accesses large data structures provides three primary advantages. First, when the thread repeats a memory access pattern (e.g., during multiple instances of a loop), we force the thread to periodically migrate between caches in the same pattern each time. As a result, the thread tends to access the same portion of the data when it is running on a specific core, resulting in lower miss rates. Second, even when the computation moves completely unpredictably through the data structures, periodic migrations result in more of the data structure residing in the combined caches. As a result, many DRAM accesses become faster (and more power efficient) cache-to-cache transfers. Finally, judicious migration while accessing very large data structures (that tend to completely over-write the cache or caches) can, in some cases, shield other data and allow it to remain in another cache.

We have developed a compiler-based, software-only data spreading system that identifies loops which have large data footprints and suitable sharing patterns (e.g., high sharing between instances of the same loop) and spreads those loops and the data they access across multiple cores, both within a chip multiprocessor or across multiple dies or sockets. Data spreading can be applied to any system, multicore or multiprocessor, with private L2 or L3 caches. Our experiments with multiple Intel and AMD multiprocessor systems show that data spreading can speed up a range of applications by an average of 17%. Most impressive, data spreading achieves this speedup without any extra power consumption. In fact, in the best case, it significantly reduces power by avoiding DRAM accesses. Finally, data spreading requires no new hardware support and, since it relies on the system's default caching behavior, does not threaten correctness.

This chapter is organized as follows. Section 3.1 describes the motivation and basic data spreading approach. Details of our experimental methodology are presented in Section 3.2. Section 3.3 describes the actual data spreading algorithm, evaluating several design options and presenting initial results. Section 3.4 examines software data spreading results more closely across different systems and working set sizes. It also examines its power efficiency and applicability to multicores. Section 3.5 concludes.

## 3.1 Software data spreading

Software data spreading allows a single thread of computation to benefit from the private cache capacity of idle cores in the system, whether on the same processor or on other, idle sockets. As the thread executes, it moves from core to core, spreading its accesses across the caches. If we time the migrations correctly, the thread can either avoid misses in the private cache it happens to be using or have its misses serviced out of another core rather than from main memory. This results in reduced execution time and energy consumption, since accesses to main memory are both slow and power-hungry.

Data spreading works best in systems with large private caches (typically L2 or L3), spread across multiple cores, dies, or sockets. As shared caches face scaling limitations, we expect private caches (or caches shared among a subset of cores) to become more common. It also applies to any machine with multiple processors on separate dies, since each die includes a private (relative to the other dies or sockets) cache.

In the next three subsections, we give some examples of how software data spreading applies in different scenarios, describe our implementation of the data spreading mechanism, and then discuss its potential impacts on performance and power efficiency.

### 3.1.1 Examples

Algorithm 1 contains a pair of loops that are good candidates for spreading. It accesses two arrays, $a$ and $b$, and we assume that the combined size of both arrays is roughly eight times the capacity of a single cache. To illustrate data spreading, we will assume (to keep the example simple) a CMP with only private caches.

**Algorithm 1 – Simple example code.** Data spreading can accelerate this code if the working set does not fit in a single L2 cache.

```
for $i = 1$ to 100 do  // Loop 0
  for $j = 1$ to 1000 do  // Loop 1
    $a_j = a_{j-1} + a_{j+1}$
  end for
  for $j = 1$ to 2000 do  // Loop 2
    $b_j = b_{j-1} + b_{j+1}$
  end for
end for
```

If this code executes on a single core, the cache miss rate will be very high,

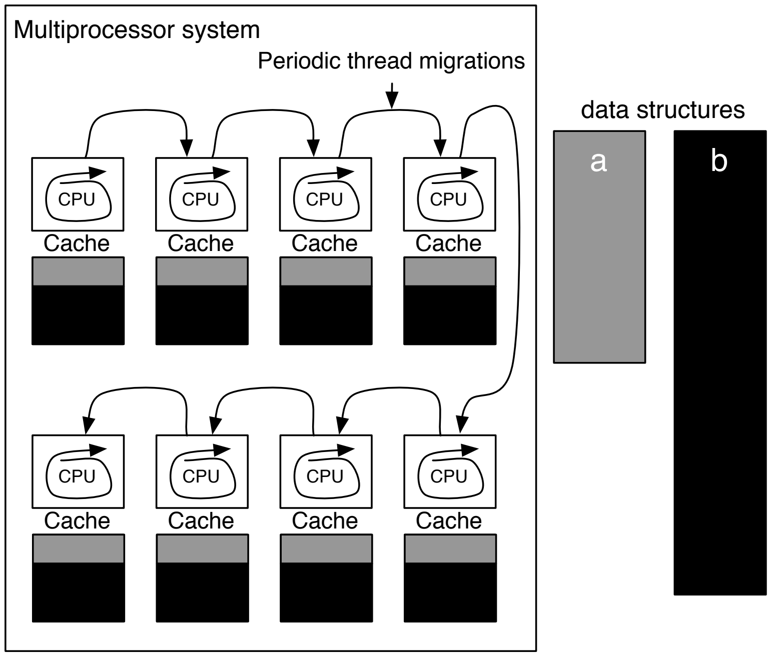


Figure 3.1: One iteration of the outer loop in Algorithm 1 with data spreading across 8 cores.

since Loops 1 and 2 will destructively interfere. However, if we have an 8-core CMP, the aggregate capacity of the cores' private caches is enough to hold both arrays, and there will be no capacity misses. Our spreading technique allows us to perform this distribution without any hardware support. Figure 3.1 illustrates the distribution of data across the private caches in the system.

Data spreading may provide benefits even if the data structures are too large to fit in the entire on-chip cache space. It will still collect a larger portion of the data into the private caches. Alternatively, we could spread as much of $b$ as will fit across all but one of the caches, and isolate the rest of it to a single cache. Accesses to the spread out portion will be fast, while the remainder will be slower. If $b$ is very large, then we can isolate execution of Loop 2 in a single core, while spreading Loop 1 to take advantage of the remaining caches. Loop 2 will "thrash" in its cache (this is unavoidable, since $b$ is large), but Loop 1 will remain largely unaffected. Our compilation system does not currently support this last option.

Software data spreading can also speed up irregular access patterns. Algo-

**Algorithm 2 – Irregular loop.** Data spreading can reduce the cost of misses for irregular access patterns.

```
for $i = 1$ to 100 do
    $p = list$
    while $p \neq null$ do
        $p = p \rightarrow next$
    end while
    Shuffle(list)
end for
```

rithm 2 traverses a linked list that is too large to fit in a single cache, then a second function shuffles the list. On a single core, most of the accesses would miss in the cache. With spreading, the number of misses to private cache remains mostly unchanged, but nearly all of them (assuming the working set fits in the combined caches) will be satisfied via cache-to-cache transfers, saving an expensive off-chip access. Current multicores do not typically support fast cache-to-cache transfers, so our experiments do not show large gains in this case; however, we expect that to change in future chips. Still, if the "shuffle" does not completely randomize the ordering, we will see gains even without fast cache-to-cache transfers.

### 3.1.2 Implementing data spreading

The only support our implementation requires is from the operating system: The OS must provide the means to "pin" a thread to a particular (new) core, and a mechanism to determine how many cores are available to the application. With this support, migrating from one core to another requires a single system call. This support already exists in most operating systems running on multicores or multiprocessors.

The main challenge in software data spreading is determining when to migrate. Our compiler profiles applications to identify the data-intensive loops it will spread. Then it adds code to count loop iterations and call the migration function periodically. Algorithm 3 shows the code from Algorithm 1 with the extra code for spreading across eight caches. We discuss the loop selection process and spreading policies in Section 3.3.

Choosing the loop's period requires balancing two opposing forces: Spreading data across as many cores as possible is desirable, since it will spread cache pressure out evenly and avoid spurious cache conflicts. However, a shorter period means additional thread migrations, which can be expensive.

**Algorithm 3** – **Data spreading in action.** The code in Algorithm 1 after the data spreading transformation.

```
for $i = 1$ to $100$ do
  for $cpu = 0$ to $7$ do
    MigrateTo(cpu)
    for $j = 125 \times cpu$ to $125 \times (cpu + 1)$ do
      $a_j = a_{j-1} + a_{j+1}$
    end for
  end for
  for $cpu = 0$ to $7$ do
    MigrateTo(cpu)
    for $j = 250 \times cpu$ to $250 \times (cpu + 1)$ do
      $b_j = b_{j-1} + b_{j+1}$
    end for
  end for
end for
```

### 3.1.3 The cost of data spreading

Like most optimizations, data spreading is not free. There are three potential costs that we must manage to make the technique profitable: its impact on the availability of other cores, the cost of thread migration, and its impact on power and energy consumption.

**Performance impact on other threads** Since data spreading increases the number of cores a thread is using, it could potentially interfere with other threads' performance. However, when idle cores are unavailable, or better used for other purposes, we can forgo data spreading – so the opportunity cost of using other cores is very low. We assume the main thread queries the OS to find the number of available cores. If it returns 0, the code runs without spreading enabled, and the only sources of overhead are the quick (it simply returns null in this case) and infrequent calls to the *Migrate_To* function.

**Thread migration cost** This is the primary cost for implementing data spreading in the systems we tested. GNU/Linux (starting with Linux kernel 2.6) provides an API to pin threads to processors, but it is an expensive operation. Linux 2.6.18 on an Intel Nehalem processor takes about $14\mu$s to perform one migration. If the other core is in a sleep state, the cost could be even higher. The high cost of migration in current systems restricts our ability to employ data spreading successfully on a single CMP, as explored further in Section 3.4.6. Proposals for hardware migration support such as [BT08] could reduce this significantly. A migration that requires OS intervention can be made to cost on the order of $2\mu$s on a 3 GHz processor with OS changes but no hardware support [SMM$^+$09].

The other cost of migration, besides the overhead of transferring the thread context itself, is cold start effects – the cost of moving frequently accessed data into the new cache, the loss of branch predictor state, BTB state, etc. Typically, cache state is the most expensive to move. Data spreading, when done correctly, minimizes this cost by moving a thread to a location where future accesses are already present in the cache (and away from a core where they are not present).

**Power cost** In contrast to traditional parallelization techniques and most of the non-traditional parallelization techniques that use multiple processing cores, data spreading can have a positive effect on both power and energy consumption. Other techniques achieve speedups by executing instructions on otherwise unused cores, and those instructions can consume extra power and energy. The only extra instructions that data spreading executes are in the migration function that moves threads between cores. More importantly, only one core is actively executing at any time.

Most current multi-core processors lack the ability to power-gate or even voltage-scale individual cores. In that case, one core being active implies they are all powered (and therefore dissipating leakage power). An idle core uses less power than a running core, but that is true whether the same core is always idle or whether activity (and, therefore, inactivity) shifts from core to core. Recent processors, such as Nehalem [neh08], are able to voltage-scale individual cores, or even power-gate cores. This will lower the power consumption of idle cores even further, strengthening the power argument for data spreading. Power gating will increase the migration latency somewhat, but when data spreading is effective, the idle periods for each core are far longer than the time required to wake the core from sleep [KTR+04, KBW12]. We can reduce this cost further by predictively waking the core several loop iterations before we plan to migrate.

Section 3.4.4 quantifies the power benefits of data spreading.

### 3.1.4 Which cores to use

The largest gains from data spreading typically come from aggregating the largest caches. For example, on a multi-socket Nehalem architecture, we gain more from aggregating L3 caches across sockets than from aggregating L2 caches on-chip. This will depend on the application – if the working set fits in four L2 caches, but not one, it will only gain from spreading at that level. Because of the large working sets of the applications we study, we focus on spreading at the socket level, except for Section 3.4.6. We also find we typically gain from using the minimum number of cores to aggregate the caches, because this reduces the frequency of migration. So for example, with two Nehalem sockets, our best gains usually come from data spreading across two cores (one on each socket), rather than across all eight cores. For the Core2Quad, we use one core per die (two per socket), and on the Opteron, we use one per socket. Tuning data spreading for individual loops and to work across multiple levels of the memory hierarchy is the subject of future work.

Table 3.1: The multiprocessor system configurations that are used to test data spreading. The systems vary in cache organizations and in access latencies.

| **System Information** | **Intel Pentium-4** | **Intel Core2Quad** | **Intel Nehalem** | **AMD Opteron** |
|---|---|---|---|---|
| CPU Model | Northwood | Harpertown | Gainestown | Opteron 2427 |
| No. of Socket × No. of Die × No. of core | 4×1×1 | 2×2×2 | 2×1×4 | 2×1×6 |
| Last Level Cache | 2M | 6M (per die) | 8M | 6M |
| Cache to cache transfer latency | 300–500 cycles | 150–250 cycles | 120–170 cycles | 210–215 cycles |
| Memory access latency | 400–500 cycles | 300–350 cycles | 200–240 cycles | 230–260 cycles |
| Migration cost | $14\mu s$ | $10\mu s$ | $14\mu s$ | $9\mu s$ |
| Linux Kernel | 2.6.9 | 2.6.28 | 2.6.18 | 2.6.29 |

## 3.2 Methodology

To evaluate our approach to data spreading, we implement it under Linux 2.6 on several real systems with Intel and AMD IA32 processors. The system configurations are given in Table 3.1. We compute the latency information for our machines by running microbenchmarks. The migration cost shown here is the latency to call the function *sched_setaffinity* that changes the CPU affinity mask, and causes thread migration. All experiments run under Linux 2.6. We use gcc 4.1.2 with optimization flag -O3 for all of our compilations, PIN 2.6 for profiling and analysis, and hardware performance counters to measure cache miss rates.

Our benchmark applications are a set of memory intensive applications from Spec2000 [Hen00], and the serial version of NAS [BBDS93]. We also pick one Spec2006 [Hen06] integer benchmark *Libquantum*, since it is easy to vary its working set size. Table 3.2 provides the name and the approximate working set size of the benchmarks. To identify the memory intensive benchmarks we use the cycle accurate simulator, SMTSIM [Tul96] (configured to roughly match one of our real experimental machines) to identify those workloads that achieve at least 75% speedup with a perfect L1 data cache. Our rationale for selecting this set of

Table 3.2: Name and resident memory requirements for benchmarks. We show both train and reference input sets.

| **Benchmark** | **Resident Memory** | **Benchmark** | **Resident Memory** |
|---|---|---|---|
| Art_T | 3 MB | BT_A | 298 MB |
| Applu_T | 20 MB | CG_A | 55 MB |
| Equake_T | 12 MB | LU_A | 45 MB |
| Mcf_T | 45 MB | MG_A | 437 MB |
| Swim_T | 56 MB | SP_A | 79 MB |
| Art_R | 4 MB | BT_B | 1200 MB |
| Applu_R | 180 MB | CG_B | 399 MB |
| Equake_R | 49 MB | LU_B | 173 MB |
| Mcf_R | 154 MB | MG_B | 437 MB |
| Swim_R | 191 MB | SP_B | 314 MB |
| Libq_R | 64 MB | | |

workloads is that if a workload shows little benefit from a perfect memory hierarchy, there is no reason to expect data spreading to offer any benefit. Furthermore, since it is a software-only technique, there is no danger of it penalizing workloads – if it is not useful for a particular application it should not be applied. Our system currently works on C code. We were able to convert Fortran77 code using an f2c converter, but we were not able to convert Fortran90 code, which excludes some of the SPEC benchmarks.

For the SPEC benchmarks, we profile using the train input, and experiment with the reference input – for some experiments we also run the train inputs, just to get more variation in working set size. When train performs better than ref, it is typically because the working set falls in the optimal range, rather than due to increased profile accuracy. For NAS, we profile using the *W* input, and experiment with both the *A* and *B* inputs. We do all the experiments multiple times (around 10). We ignore some of the outliers and use the average execution time as our result. The variation of execution times usually stay within 5%. The results presented here are all normalized with respect to unmodified code.

## 3.3 Data Spreading in the Compiler

Our software data spreading compilation system involves a profiling step, a loop identification and selection stage, and a loop transformation step. This section describes and evaluates options for all three steps.

### 3.3.1 Profile collection and program structure analysis

Our system uses PIN [LCM+05] to statically identify loops, profile the program, and collect information about all the loops that make up a program's execution. Each time a loop executes, an event we term a loop *epoch*, we count the number of iterations it executes, the set of cache lines it accesses, and the number of memory accesses it makes each iteration. The profiling part takes 50 times that of normal execution time for a program.

We profile unoptimized code to simplify the process of identifying loops in the binary and connecting them with the source code. This is necessary as the compiler does optimizations like loop unrolling and function inlining. However, with some compiler support, profiling on optimized code would be possible. After we transform loops based on this analysis, full optimization is applied to generate code for our measurements.

For each loop, $\ell$, we calculate its *total memory footprint* as the set of cache lines $\ell$ touches across all its epochs and its *epoch footprint* as the lines $\ell$ touches during the execution of a single epoch. We also define $M_\ell$ to be the maximum footprint size for any epoch of $\ell$ and $N_l$ to be the number of iterations in that epoch.

We use this data to compute the *epoch sharing ratio* (ES) for each loop. Intuitively, epoch sharing is a measure of the degree to which multiple epochs of the same loop touch the same data. Loops with large ES values are good candidates for spreading, because once the first epoch fetches the epoch-shared data into the caches, the following epochs will likely reap significant benefit when they access the same data.

To compute the ES for a loop, we start by calculating, for each loop epoch,

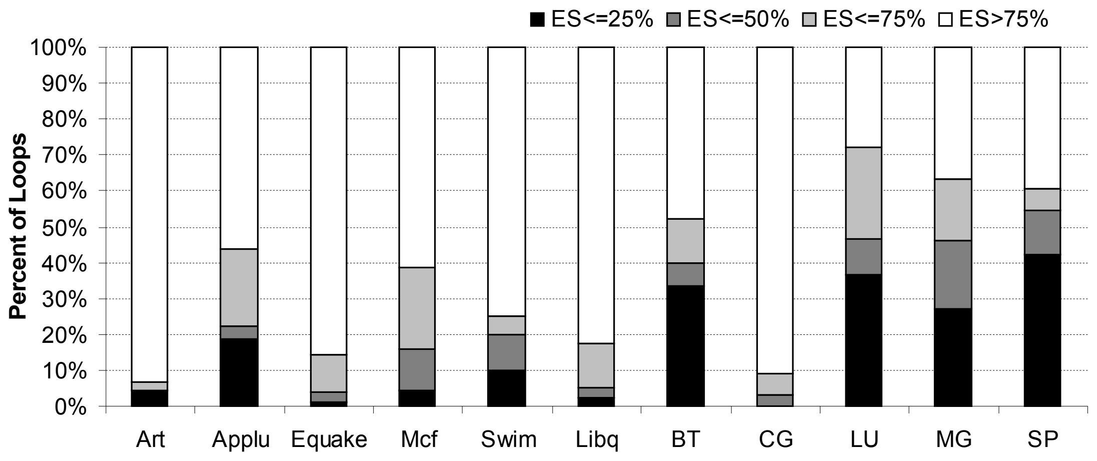


Figure 3.2: Epoch sharing breakdown of data intensive applications. Data spreading is effective for loops with high epoch sharing.

the fraction of that epoch's footprint that overlaps with the union of the footprints of all previous epochs of the same loop. The final ES is the average of this value over all the epochs of the loop. Formally, if $\ell$ has $k$ epochs and the corresponding epoch footprints are $e_0, e_1, \ldots, e_{k-1}$, the epoch sharing of $\ell$ is

$$ES(\ell) = \frac{\sum_{i=0}^{k-1} S(\bigcup_{j<i} e_j, e_i)}{k-1}.$$

where

$$S(e_a, e_b) = \frac{|e_a \cap e_b| \times 100}{|e_a \cup e_b|}$$

Here $S(e_a, e_b)$ denotes the sharing between two epochs $a$ and $b$ with footprints $e_a$ and $e_b$.

For example, in Algorithm 1, loops 1 and 2 each have epoch sharing of 100, since they always touch the same data. Loops with only one epoch have an epoch sharing of zero in our analysis. We find epoch sharing to be a very common characteristic of loops in data intensive applications. In Figure 3.2, we compute the epoch sharing breakdown of loops with more than 1 epoch. It clearly shows that loops reuse data heavily. Even for an irregular benchmark like *mcf*, we see that 61% of loops have epoch sharing of more than 75%.

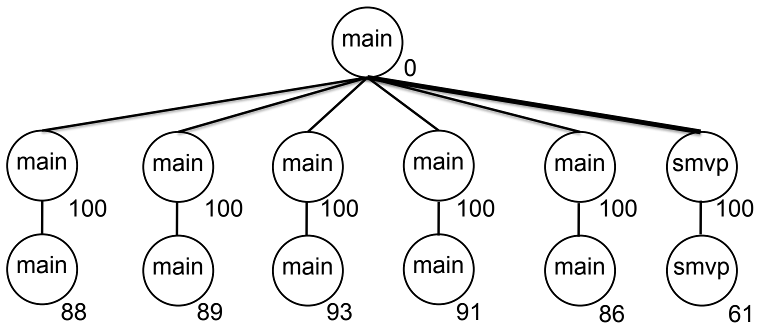


Figure 3.3: Loop nest tree annotated with epoch sharing values for the main kernel of *equake*.

We use the profile data to create a dynamic loop nesting tree for the application. Each node in the tree represents a single loop, and its children are the loops nested within it. We ignore function call boundaries when creating the tree (i.e., if a loop calls a function that contains loops, those loops are directly nested within the calling loop). Our analysis does not currently handle recursion, so we ignore back edges in the loop nest tree. Figure 3.3 shows the loop nest tree of *equake* from Spec2000 [Hen00], annotated with epoch sharing values.

Clearly, data spreading is most effective on loops with large memory footprints and high epoch sharing. If an inner loop (child loop) has high epoch sharing, it is likely that the outer (parent) loop also exhibits high epoch sharing. However, it does not work to spread both loops, as the migrations in the inner loop will just override the outer loop migration commands. For example, the outer loop in Algorithm 1 inherits all of its epoch sharing from the inner loops, and spreading the outer loop would actually hurt performance. Even when both parent and child loop have significant epoch sharing, and the parent does not inherit the sharing from the child, the child is often a better candidate to spread (assuming its working set exceeds the cache size). Currently, we always spread the child in this situation.

## 3.3.2 Baseline spreading algorithm

Our baseline algorithm takes three parameters to do loop selection – an epoch sharing threshold, $E_{\text{min}}$, a minimum footprint size, $F_{\text{min}}$, and a maximum

Table 3.3: Description of different loop selection policies. The policies use different threshold values for epoch footprint and ES.

| **Policy** | **Description** |
|---|---|
| FP16 | Epoch footprint $\geq$ 16KB |
| FP32 | Epoch footprint $\geq$ 32KB |
| FP64 | Epoch footprint $\geq$ 64KB |
| ES25-FP32 | Epoch footprint $\geq$ 32KB and ES $\geq$25% |
| ES75-FP32 | Epoch footprint $\geq$ 32KB and ES $\geq$75% |
| ES25-FP32-MG | Epoch footprint $\geq$ 32KB and ES $\geq$25% and Epochs/sec $\leq$ 1000 |

migration frequency. We examined other inputs, such as sharing with siblings, etc., but found that our best algorithms used just these three. The algorithm examines the nodes of the loop nest graph in reverse topological order, starting at the leaves and working toward the root. The algorithm selects a loop for spreading if a) none of its descendants have been selected and b) its epoch sharing and epoch footprints are larger than $E_{\text{min}}$ and $F_{\text{min}}$, respectively. Typically, $F_{\text{min}}$ would be set to a value smaller than the cache, given that the profiled working set is not necessarily indicative of the working set size of future runs.

Once the candidate loops are finalized, we do simple source to source transformations. For loops with known iteration bounds before entering the loop, we spread it using Algorithm 3. For loops with unknown iteration bounds, we need to know the expected iteration count – $N_l$ from the profile.

### 3.3.3 Candidate loop selection

The basic algorithm described above provides three parameters that we can tune to improve performance. Table 3.3 lists the settings we evaluate. Table 3.4 gives the number of selected loops for each policy on our benchmarks, and shows how loops are filtered across different policies. For instance, FP16 admits just 24% of all loops, indicating that there are a significant number of loops with very small footprints. Policy ES25-FP32-MG is more restrictive, and includes only 15% of all static loops.

Table 3.4: Number of selected loops for different selection policies. The threshold on epoch footprint filters more than 75% of loops.

| **Bench mark** | **Total Loops** | **FP16** | **FP32** | **FP64** | **ES25 FP32** | **ES75 FP32** | **ES25 FP32 MG** |
|---|---|---|---|---|---|---|---|
| Art | 84 | 32 | 32 | 31 | 20 | 20 | 20 |
| Applu | 199 | 23 | 23 | 19 | 12 | 11 | 12 |
| Equake | 121 | 27 | 27 | 27 | 6 | 6 | 6 |
| Mcf | 70 | 22 | 19 | 19 | 11 | 7 | 9 |
| Swim | 69 | 16 | 14 | 12 | 9 | 9 | 9 |
| Libq | 63 | 16 | 16 | 16 | 14 | 9 | 14 |
| BT | 243 | 56 | 52 | 50 | 43 | 43 | 43 |
| CG | 63 | 30 | 26 | 20 | 14 | 14 | 14 |
| LU | 190 | 41 | 35 | 27 | 15 | 15 | 15 |
| MG | 82 | 15 | 15 | 15 | 8 | 5 | 6 |
| SP | 333 | 82 | 82 | 76 | 72 | 72 | 72 |

Figure 3.4 shows the performance improvement for all six policies shown in Table 3.3, run on a 2-socket (4 die) Core2Quad machine. The first three policies filter loops based solely on footprint size. Allowing loops with very small footprints (<16KB) degrades performance by 18%, but performance improves by 3% on average if we require footprints to be at least 64KB.

The next three policies use the epoch sharing threshold to filter loops that do not benefit from data spreading because they touch different data during each epoch. Although without ES considerations, the FP64 limit outperformed FP32, we find that using a more liberal policy (FP32) and then letting the ES restriction pare down the list was preferable. Limiting epoch sharing to at least 25% works well, and the ES25-FP32 policy provides a small speedup (compared to a slowdown for FP32). Increasing the ES limit to 75% reduces the slowdown of two benchmarks– *Swim* and *MG* by filtering some loops that cause too frequent migrations. The last policy, which also guards against too-frequent migrations, improves performance further (up to an 8% speedup), in large part by eliminating slowdowns where data spreading does not work well. This policy eliminates loops whose epochs occur more than once per millisecond. This caps migration overhead

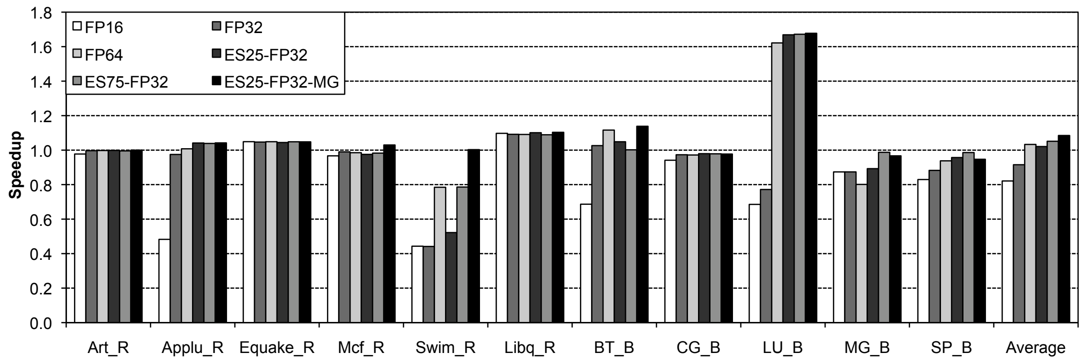


Figure 3.4: Wall clock speedup across all benchmarks using six different policies in the Core2Quad system.

at 5% of execution time. Ultimately, we see that a combination of large footprint, high ES sharing and avoiding too-frequent migrations yields the best candidates for spreading.

*LU* clearly gains the most from software data spreading. As we explore in Section 3.4, performance is highly sensitive to working set size. Unfortunately, *LU* is the only one of these applications with a significant data structure in the sweet spot – between the size of one and four caches. But we also get reasonable gains on *BT* and *libquantum*.

*Swim* and *Mcf* each see little or no benefit for any of the policies. *Swim* has a very large working set of around 191MB, so even using four caches (combined cache space 24MB) data spreading fails to keep sufficient useful data. In the case of *Mcf*, most of the data accesses are irregular, and we do not see benefit because of expensive cache-to-cache transfer in our real experimental system.

For current systems, migration overhead has a significant impact on the performance of data spreading. The split of user vs. kernel time provides insight into these costs. For policy FP16, the most aggressive, *LU* sees a 14% user time speedup, but the increase in kernel time due to OS migration code resulted in a wallclock slowdown of 31%. This implies techniques that reduce migration costs [SMM$^{+}$09, BT08] will increase the benefits of data spreading and compilers can apply the technique more aggressively. Section 3.4.5 explores this further.

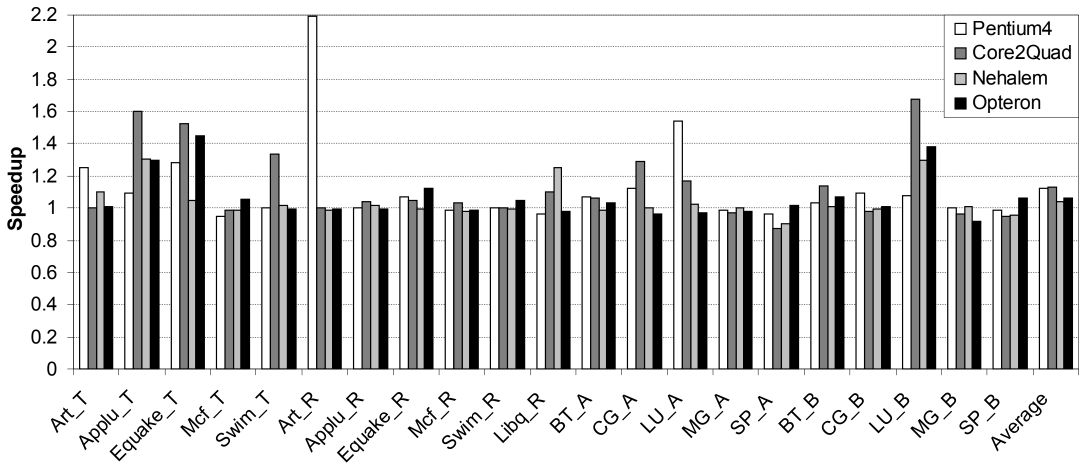


Figure 3.5: Speedup across different machines using the ES25-FP32-MG policy. The improvement can be as high as 2.2×.

### 3.3.4 Loop spreading policies

Once we have selected a loop for spreading, we must determine how and when to migrate. We considered several different policies, but the results in this chapter only reflect one. That policy is also our simplest.

The *balanced* policy spreads loop iterations evenly across the available cores. Our results for more complex policies failed to show significant or consistent gains. For example, policies that considered the size of the cache more explicitly tended to reduce sharing between sibling loops. For example, the first one quarter of loop $A$ tended to touch the same data as the first one quarter of loop $B$. With *balanced*, those data all go into the same cache. With other policies, we might migrate at different points in the two loops depending on how much other data was being touched. The results shown in this chapter all use the balanced policy.

## 3.4 Understanding Data Spreading

This section strives to acquire a deeper understanding of data spreading, particularly in light of the uneven performance gains demonstrated in the previous section. It uses microbenchmarks, kernels, and one full benchmark running on a whole suite of real machines to examine the interplay of working set size and data spreading effectiveness. It also measures data spreading's impact on power and energy. Finally, it examines data spreading's sensitivity to migration overhead, especially for spreading within a CMP.

### 3.4.1 Diverse memory hierarchies

Results for our benchmark suite on four different machine architectures are shown in Figure 3.5. Here we see that each machine gets speedup from data spreading, but the speedups are not very predictable. In most cases, speedup is simply a case of whether or not the key data structures fit in the aggregated caches. Since each of these architectures has a different cache hierarchy, the results varied. We see, though, that across a diverse range of working sets, performance improved overall. Both the Pentium 4 and the Core2Quad systems achieve an average speedup of 13%.

### 3.4.2 Microbenchmarks

To further understand the sensitivity of these techniques to working set size, we focus on a more restrictive set of benchmarks that give us the ability to modify the working set size continuously. First, we create two microbenchmarks – one accesses data sequentially, the other chases pointers through memory at random. Both perform the same set of accesses on each iteration through an outer loop. We ran these benchmarks on four different machines, as shown in Figure 3.6. The Core2Quad results (Figure 3.6b) illustrate the phenomenon well. Without data spreading, when the working set fits in the cache, throughput is high, but it degrades quickly when the working set overflows the cache. With data spreading, there is a cost when the data fit in a single cache, and it asymptotically matches the baseline when the working set is very large. But in between, data spreading *significantly* extends the region where we maintain close to full throughput. This is true both for sequential access (where the hardware prefetcher is actively assisting performance) and random access (where it is not). All four machines show a similar

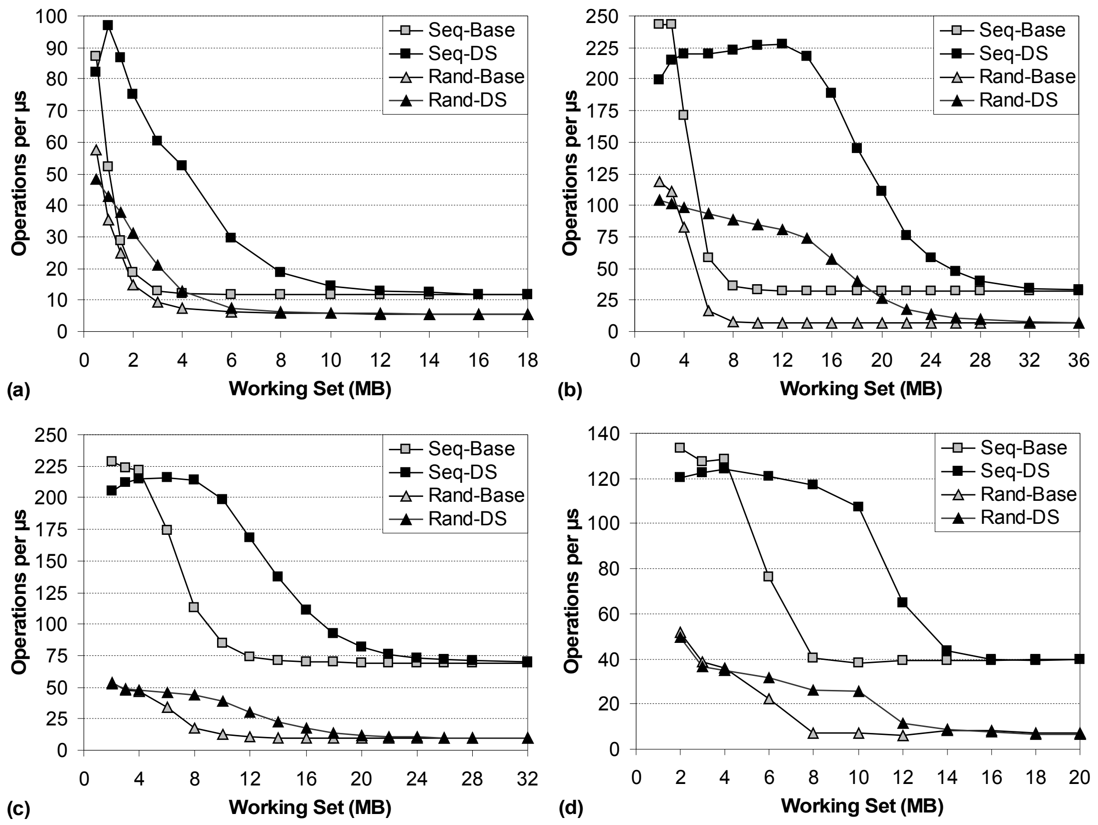


Figure 3.6: Data spreading throughput for sequential and random access for different machines – (a) Pentium 4, (b) Core2Quad, (c) Nehalem, (d) Opteron. The '-Base' data are for code that does not perform data spreading.

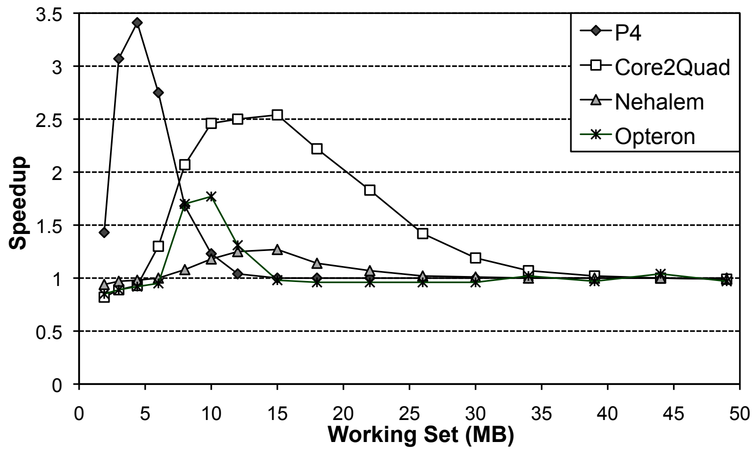


Figure 3.7: Speedup across different machines for 2D Jacobi using data spreading.

effect.

Also notice that the data spreading curves do not drop as sharply as the baseline – we continue to get some performance even after the working set no longer fits fully in the aggregated caches.

### 3.4.3 Applications

We can perform similar experiments for two full applications with easily configurable working sets – the 2D Jacobi kernel and the SPEC2006 libquantum application. For these, we show speedup of data spreading over the baseline for a given working set size.

The Jacobi results are shown in Figure 3.7. The graph shows speedup for the region between the size of a single cache and the size of the aggregated caches that data spreading operates on. On either side of this "hump" data spreading offers no benefits. All the machines achieve speedups between 1.25 (Nehalem) and 3.5 (Pentium 4). The Pentium 4 does especially well because it has the highest latency to main memory. The Core2Quad delivers improvements over the widest range – from 5 to 35 MB.

Results for *libquantum* from the SPEC2006 Integer benchmark suite are in Figure 3.8. Libquantum is a more complex computation with multiple, and varied,

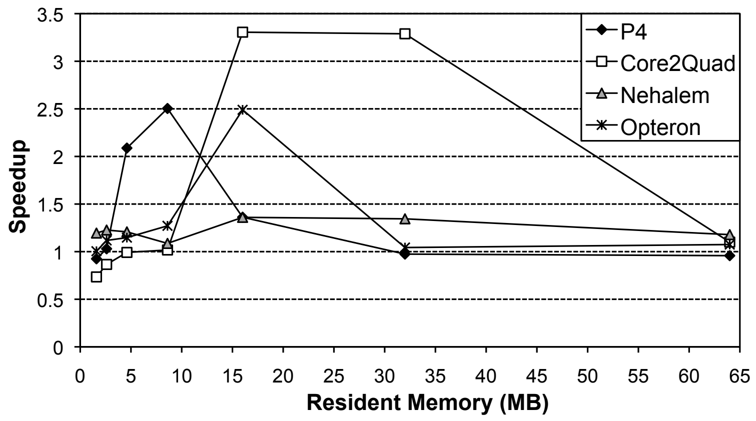


Figure 3.8: Speedup across different machines for Libquantum using data spreading.

spread loops. Although the complexity of the application adds some noise to the results, the same trends emerge: Data spreading provides speedup after the local cache is exhausted, again extending the high-IPC range of the benchmark across a wider range of input sizes. The Core2Quad achieves over 3× speedup for this application.

These results all point to the same conclusion. While data spreading does not always provide speedups over non-spread code, it consistently makes the application's performance *significantly more robust* in the face of varying working set size. In the best case, data spreading achieves dramatic speedups – it achieves parallel-type speedups on parallel machines, without ever requiring parallel execution. This last point makes the optimization particularly attractive from an energy efficiency standpoint, which is explored further in the next section.

## 3.4.4 Power

On the surface, using multiple cores to execute a single thread does not appear to be a power-conserving optimization. However, just the opposite is true, since only one core is ever active at any time. Indeed, data spreading can *save* power (and energy) by eliminating accesses to DRAM.

We use a power meter on two of our experimental systems to quantify this

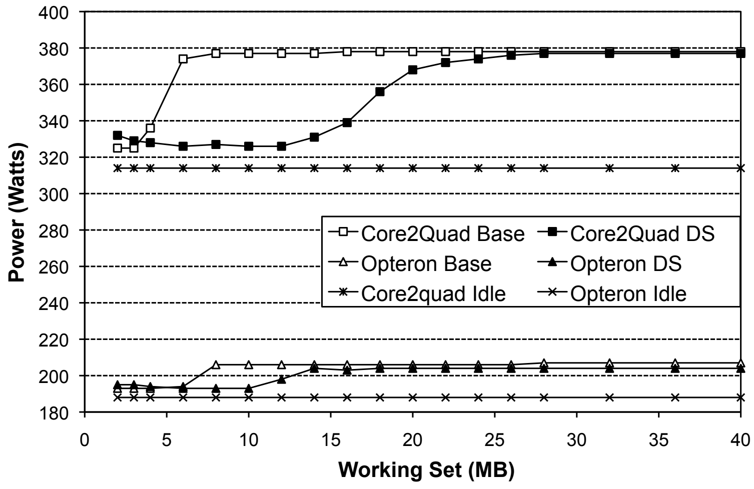


Figure 3.9: Power requirements for random access in Core2Quad and Opteron. The idle power shows the power consumption when no workload is running.

effect for our random access microbenchmark. We measure the total power of the entire system. So, it reports the power consumption by all system components including the memory system, cooling system, storage devices, network cards, etc. Figure 3.9 shows the results. Where data spreading is effective, the power savings is dramatic. Spreading on the Core2quad saves up to 51W, or 81% of non-idle system power. For the Opteron, the savings are smaller – 13W, or 72% of non-idle power. The results also show the cost of the unnecessary migrations: At 2MB, migrations increase power consumption by up to 7W and 2W on the Core2Quad and Opteron, respectively.

From this graph, we see the dramatic impact on power when DRAM accesses are made unnecessary. This bodes well for our real applications, where DRAM accesses are consistently reduced, in some cases by an order of magnitude. Figure 3.10 shows the normalized (to no data spreading) last level cache (LLC) misses for the Core2Quad system. These results correlate well with the speedup results in Figure 3.5. Note that spreading-induced coherence (cache-to-cache) transfers show up as misses, even though in some machines they will be faster (and lower power) than memory misses. Data spreading converts 80 and 90% of misses into local hits for *LU* (B input) and *applu* (train input), respec-

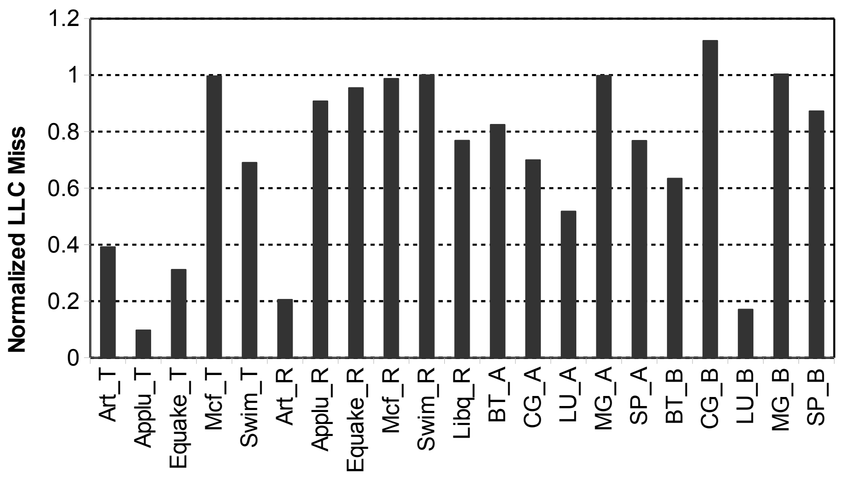


Figure 3.10: Reduction of last level cache misses for Core2Quad using data spreading.

tively. For 11 of our 21 benchmarks, data spreading eliminates 20% of misses or more.

### 3.4.5 Migration overhead

Data spreading's primary overhead comes from the migrations it requires, especially the software overhead. Triggering these switches with the system call, *sched_setaffinity* incurs a fixed overhead of 9 to 14 $\mu$s (Table 3.1), since it requires a trap to the operating system. If context switches occur too frequently, this overhead will eliminate the benefits of data spreading. Increasing the number of cores exacerbates this problems, since the frequency of context switching increases linearly with the core count. This is unfortunate, since spreading across more cores also increases the amount of cache capacity that data spreading can exploit.

To reduce the migration overhead, we developed a userspace context switch mechanism (*User-CS* in the figures, as opposed to *OS-CS*) that uses user space migrations (via `setcontext()` and `getcontext()`) and spin locks to reduce migration costs to just 1-3$\mu$s. Idle cores spin until called upon to wake up and acquire the context of the running thread.

Figure 3.11 compares the performance of both migration schemes for our

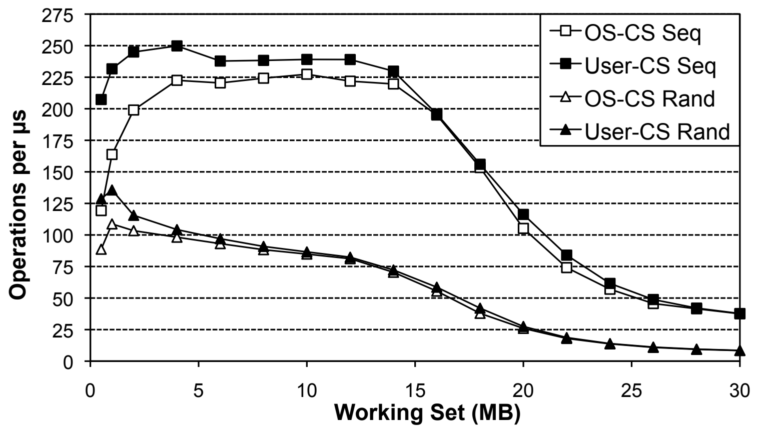


Figure 3.11: Impact of userspace and OS directed thread migration in Core2Quad for the microbenchmarks.

microbenchmarks on the Core2Quad. The User-CS scheme increases performance by 5% for working set sizes up to 10 MB (12% for 4 MB or less) for sequential accesses and by up to 2% for working set sizes under 12 MB (6% 4 MB or less) for random accesses. The gain is larger for smaller working sets because migration occurs more frequently in this case.

Figure 3.12 shows the comparison between User-CS and OS-CS for the full benchmark suite on the Core2Quad system. Overall, User-CS gives us a 17% speedup on average (compared to 13% average speedup by OS-CS). Again, user-CS provides the greatest new boost for small working sets (like the train input sets); it also reduces or eliminates performance degradation we previously observed when we were experiencing frequent migrations (e.g., SP).

The P4 machine also sees 4% additional speedup when we use user-level migration. However, we do not notice significant changes for Opteron or Nehalem. In these machines, we use two cores (vs. 4 cores used in Core2Quad and P4) which reduces the frequency of migrations by half.

User-level migration, as demonstrated here, is a reasonable approach when performance is the highest goal. However, the idle, spinning threads may reduce the power savings that data spreading can deliver.

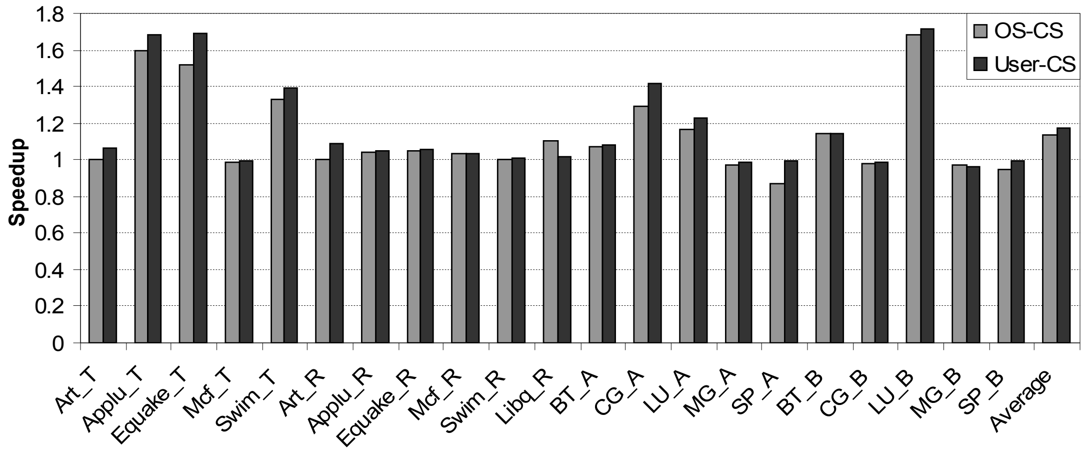


Figure 3.12: Speedup in Core2Quad using user space thread migration. User-CS provides 4% additional speedup on average.

### 3.4.6 Data spreading in CMPs

So far we have applied data spreading across sockets and dies, but data spreading can also be applied in multicore architectures. However the smaller private caches in these architectures mean that working sets small enough to fit into the aggregate on-chip private caches will induce frequent migrations. Therefore, data spreading requires the use of a fast migration mechanism (like User-CS) to realize performance gains.

To evaluate CMP data spreading we ran experiments on the Nehalem (4 core) and Opteron (6 core) machines. Nehalem has 256K private L2 caches per core whereas the Opteron has 512K private L2 caches. Figure 3.13 shows the results for the microbenchmarks on both machines. The combined cache space in Nehalem is 1M, so the benefit comes when the working set lies between 300K and 1M. The combined cache space in Opteron is 3M and so data spreading improves performance over a broader region.

Figure 3.14 shows the performance improvement for different benchmarks on Opteron while spreading is deployed within the single socket. Overall we see 6% average performance improvement and, as expected, the speedup mainly occurs for smaller working sets.

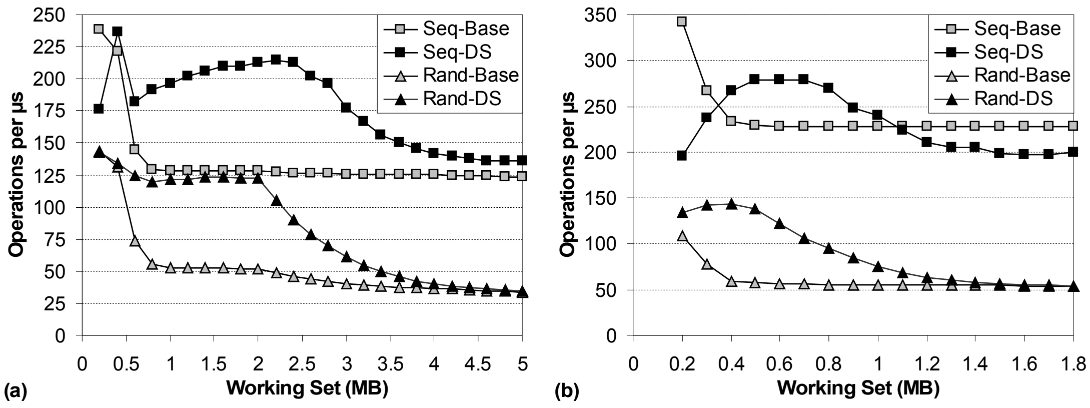


Figure 3.13: CMP data spreading throughput for sequential and random access on – (a) 6-core Opteron, (b) 4-core Nehalem.

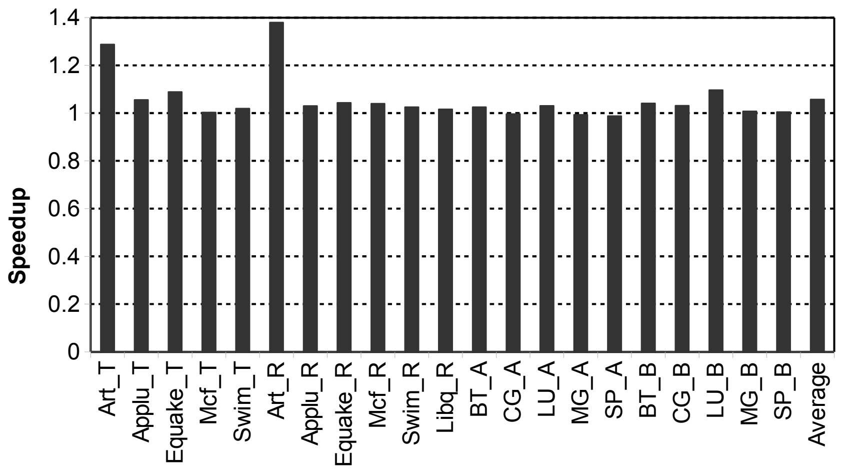


Figure 3.14: Data spreading performance on a single 6-core Opteron CMP. Art sees 38% speedup for the reference input.

## 3.5 Conclusion

This chapter presents a new compiler optimization, software data spreading, targeted at multiprocessors and multicore processors. It uses thread migration to allow a single thread to utilize the space of multiple private caches. This allows the program to transform off-chip accesses into local cache hits when the data access pattern is highly repetitious. In the case where it is not predictably repeatable, it still turns DRAM accesses into cache-to-cache transfers. Using an approach that relies on profiling to identify loops with large data footprints and to characterize their sharing patterns, we identify for each application a small set of loops that are spread across multiple caches via migration. We achieve average speedups of 17% using four processors. Speedups on the SPEC2006 libquantum benchmark are as high as 3.3x, depending on the input size. Data spreading can also provide significant power and energy savings since it actively uses only one core at a time and can dramatically reduce memory accesses.

## Acknowledgments

Portions of this research were funded by the US National Science Foundation grants CCF-0702349 and NSF-0643880.

This chapter contains material from “Software Data Spreading: Leveraging Distributed Caches to Improve Single Thread Performance”, by Md Kamruzzaman, Steven Swanson and Dean M. Tullsen, which appears in PLDI’10: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation. The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM). Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page in print or the first screen in digital media. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is

# Chapter 4

# Inter-core Prefetching: Exploiting Migrating Helper Threads

In the previous chapter, we demonstrate how non-traditional parallelism like data spreading allows programs with limited parallelism to profitably use several cores per thread, increasing the application's scalability. However, data spreading does not provide benefit if the working set is too large or the program does not have a repetitive access pattern. In this chapter, we introduce a "helper thread" based software prefetching technique called inter-core prefetching that removes these two limitations.

Prior work [CSK+99, CWT+01, CTWS01, ZS01, Luk01, KY02, KLW+04, LWW+02] has demonstrated the use of helper threads which run concurrently in separate contexts of a simultaneous multithreading (SMT) processor and significantly improve single thread performance. Helper thread prefetching has advantages over traditional prefetching mechanisms – it can follow more complex patterns than either hardware prefetchers or in-code software prefetchers, and it does not stop when the main thread stalls, since the prefetch threads and the main thread are not coupled together. However, SMT prefetching has its limitations. The helper thread competes with the main thread for pipeline resources, cache bandwidth, TLB entries, and even cache space. The helper thread cannot target more distant levels of the cache hierarchy without thrashing in the L1 cache. In addition, core parallelism is typically more abundant than thread parallelism –

even on a 4-core Nehalem, only 1 SMT thread is available for prefetching. Perhaps most importantly, not all multicores support SMT.

In contrast to prior approaches, inter-core prefetching lets helper threads run on separate cores of a multicore and/or multi-socket computer system. Multiple threads running on separate cores can significantly improve overall performance by aggressively prefetching data into the cache of one core while the main thread executes on another core. When the prefetcher has prefetched a cache's worth of data, it moves on to another core and continues prefetching. Meanwhile, when the main thread arrives at the point where it will access the prefetched data, it migrates to the core that the prefetcher recently vacated. It arrives and finds most of the data it will need is waiting for it, and memory accesses that would have been misses to main memory become cache hits.

Inter-core prefetching can target both distant shared caches and caches private to the core. Inter-core prefetching is especially attractive because it works with currently available hardware and system software. Thus, it would be easy to incorporate in a compiler or runtime system. Previous approaches have required fundamental changes to the microarchitecture or were limited to prefetching into shared caches.

This chapter describes inter-core prefetching and our implementation under Linux. We characterize the potential impact of inter-core prefetching on a range of currently-available multicore processors running focused microbenchmarks and then demonstrate its impact on a wide range of memory-intensive applications. Our results show that inter-core prefetching improves performance by an average of 31 to 63%, depending on the architecture, and speeds up some applications by as much as 2.8×. We also demonstrate that inter-core prefetching reduces energy consumption by between 11 and 26% on average. Finally, we show that because of the separation of resources, inter-core prefetching is more effective than SMT thread prefetching.

The remainder of this chapter is organized as follows: Section 4.1 describes the inter-core prefetching technique in detail. Sections 4.2 and 4.3 contain our methodology and results, and Section 4.4 concludes.

## 4.1 Inter-core Prefetching

Inter-core prefetching allows a program to use multiple processor cores to accelerate a single thread of execution. The program uses one to perform the computation (i.e., the *main thread*). The other cores run *prefetching threads*, which prefetch data for the main thread. The two types of threads migrate between cores with the prefetching threads leading the way. Thus, when the main thread arrives on a core, much of the data it needs is waiting for it. The result is reduced average memory access time for the main thread and an increase in overall performance. Inter-core prefetching is a software-only technique and requires no special support from the processor's instruction set, the operating system, caches, or coherence.

Inter-core prefetching works by dividing program execution into *chunks*. A chunk often corresponds to a set of loop iterations, but chunk boundaries can occur anywhere in a program's execution. The *chunk size* describes the memory footprint of a chunk. So, a 256 KB chunk might correspond to a set of loop iterations that will access 256 KB of data. For example, if a loop touches 1 MB of data in 100 iterations, then a 256 KB chunk would consist of 25 iterations, a 512 KB chunk would consist of 50 iterations.

Both the main thread and prefetch thread execute one chunk at a time. In the main thread, the chunked code is very similar to the original code, except for the calls to the run-time that implement migration.

The prefetch thread executes a *distilled* [ZS01, CWT+01, KY02, QMS+05] version of the main thread, with just enough code to compute addresses and bring the necessary data into the cache. We will use the term prefetch slice, or *p-slice* (terminology borrowed from prior work) to describe the code that the prefetch thread executes to prefetch the data.

Figure 4.1 illustrates inter-core prefetching with one main thread (solid line) and one prefetcher thread (dotted line) executing on two cores. Execution begins with the main thread executing chunk 1 and the prefetcher executing chunk 2. Once they complete their chunks, they swap cores and the main thread starts on chunk 2 while the prefetcher moves on to chunk 3.

Sometimes, multiple prefetch threads are useful. In a four-core system,

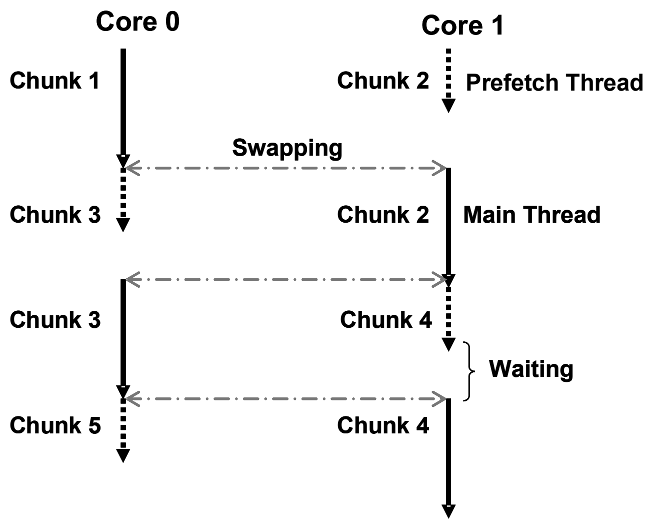


Figure 4.1: Program execution path for inter-core prefetching. The main thread and the prefetch thread swap cores repeatedly, when the main thread reaches a new chunk of data.

three cores would prefetch chunks 2-4 while the main thread executes chunk 1. When the main thread starts executing chunk 2, the prefetch thread for chunk 2 would move on to chunk 5. In this scenario, each prefetcher thread has 3 chunks worth of time to finish prefetching the chunk.

Our implementation of inter-core prefetching comprises three principal components. First, a program analysis algorithm identifies chunk boundaries in the original program. Second, a p-slice generator creates a distilled version of the code for the chunks. Finally, the inter-core prefetching library provides a mechanism to keep the main thread and the prefetcher threads synchronized as they migrate between cores.

Below we describe the key components of inter-core prefetching in detail and present an example that illustrates inter-core prefetching in action.

### 4.1.1 Identifying chunks

A chunk is a sequence of executed instructions which access a portion of the application's address space that matches (as closely as possible) the target chunk size. We use two different approaches for partitioning the iteration space into chunks – *aligned chunks* and *unaligned chunks*.

Whenever possible, we use aligned chunks. Aligned chunks force chunk boundaries to align with loop iteration boundaries at some level of the loop nest. For instance, if an inner loop accesses 100 KB and the target chunk size is 256 KB, we will likely start and end a chunk every two iterations of the outer loop.

There are some situations where inner loop iterations follow predictable patterns even though the outer loop does not. In that case, it is easier to add a counter and track inner loop iterations, ignoring the structure of the outer loop – that is an unaligned chunk. Unaligned chunks allow us to swap after a predictable number of inner loop iterations, even if we are not sure how many outer loop iterations will have executed. For example, if we are targeting 1024 inner loop iterations, and the inner loop executes 200, 700, then 300 iterations, the first swap would happen in the middle of the outer loop's 3rd iteration. For some loops, it may not be possible to predict the future addresses at all – pointer chasing code being a prime example. We do not prefetch those loops.

We prefer aligned chunks, even when there is not a precise match between inner loop sizes and the desired chunk size. They introduce less overhead because they allow us to use the existing structure of the code to introduce the thread management primitives. In practice, unaligned chunks are rarely needed.

To identify useful chunks, a profiler selects loops that perform at least 0.5% of the total memory accesses. The profiler also creates a dynamic loop nest tree that spans procedure boundaries, and calculates the average memory footprint of an iteration in each loop. While we will invariably suffer from profile mismatch, the information on the data touched in a single iteration is reliable enough to be useful across multiple inputs.

### 4.1.2 Distilling code

To build the prefetching thread, we generate p-slices by hand, following the example of much of the original work on helper thread prefetching [ZS01, CWT$^+$01, Luk01]. Generating aligned chunks is typically simple and the techniques we apply are all implementable in a compiler. Other efforts have used both static [KY02, QMS$^+$05] and dynamic [ZTC07] compilation, as well as specialized hardware [CTWS01] to construct p-slices. Those techniques could be applied to inter-core prefetching as well.

For both chunk types (aligned and unaligned), the p-slice contains just the code necessary to compute the prefetch addresses and perform the prefetches. In most cases, the distilled code matches the basic structure of the main thread, but this is not necessary. For instance, if the compiler can determine the range of addresses the code will access, it can easily generate a generic prefetcher to load the data with minimal overhead — unlike SMT prefetchers, the ordering of the accesses within a chunk is unimportant in our scheme.

Our implementation uses normal loads rather than prefetch instructions, because the hardware has the option of ignoring prefetch instructions if there is contention for memory. Using normal loads requires that the prefetcher only issues memory requests to valid addresses. This was not a problem for any of the workloads we examined, but a more aggressive implementation that issued speculative prefetch instructions to potentially invalid addresses could use prefetch instructions to avoid segmentation faults.

Our p-slices include several other optimizations as well: We convert stores into loads to prevent misspeculation and avoid invalidations that could delay the main thread. We use profile information to determine the most likely direction for branches within a loop iteration. For highly biased branches, we remove the unlikely path and the instructions that compute the branch condition.

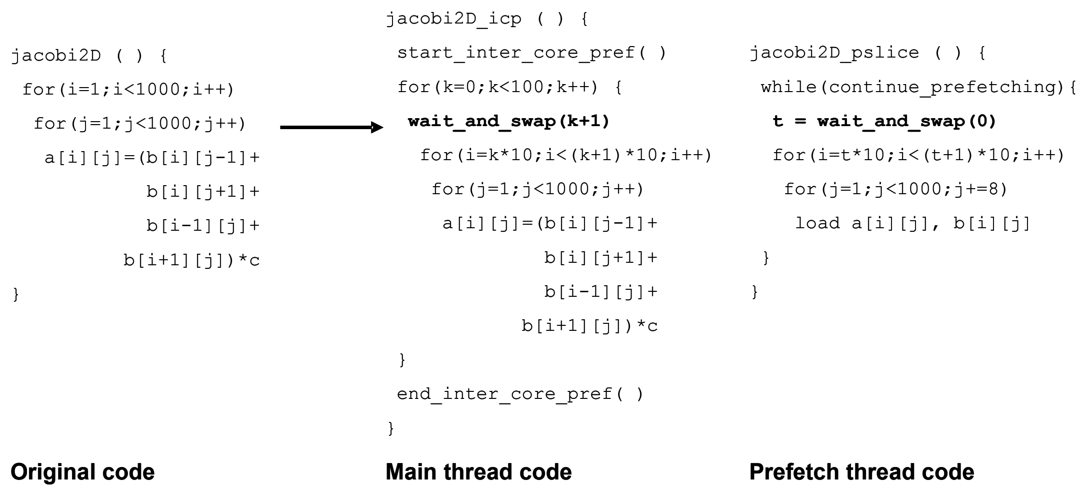
### 4.1.3 Thread coordination

For inter-core prefetching to work properly, the main thread and prefetcher threads must work together smoothly. This means that (1) they must take the same path through the program so they execute the same chunks in the same order, (2) the prefetch thread must stay far enough of the main thread to make the prefetches useful, and (3) they need to synchronize at swap points to exchange cores.

The first requirement is part of creating valid p-slices: The distiller must ensure that both the prefetcher and the main thread take the same route through the program. To handle the second requirement, we always start a prefetch thread at least a full chunk ahead. The prefetch thread can finish its chunk either earlier or later than the main thread does. The prefetcher usually finishes first, in which case one prefetching thread is sufficient (as in Figure 4.1) and the main thread will rarely stall. In some cases, the p-slice is slower than the main thread, and the main thread must wait for the prefetcher to finish. When that happens, we start prefetching the next chunk of data in the main thread's core to avoid wasting cycles, but we would prefer to never have to stall the main thread. As an alternative, we could stop the prefetching thread prematurely and allow the main thread to execute without any stall. We experiment with this approach but do not observe any advantage in our experiments. The reason is that if the main thread finishes early, eventually either the main thread or the prefetch thread will have to bring in the missing data for the next chunk to execute. Often the prefetcher (because it has more memory level parallelism and less overhead) will do so faster than the main thread, so it is better to just let it finish.

As a third alternative, very effective when we expect the prefetch threads to consistently be slower, we can use multiple prefetching threads to reduce or eliminate the need for the main thread to stall, and increase the effectiveness of prefetching. This exploits additional parallelism to allow the prefetch threads to stay ahead.

To coordinate thread migration, we have implemented a simple user-space migration tool that allows threads to migrate between cores without entering the kernel. This takes around 1–2$\mu$s and saves significant overhead since normal context switches take about 10$\mu$s on our machines. At each wait and swap point, the distiller inserts a call to a library function in both the main thread and the prefetching thread. The function acts as a barrier across the main thread and the prefetching thread working on the next chunk the main thread will execute, so the threads block until they have both completed their chunk. Then the threads swap cores, and the main thread starts executing on the core with a freshly populated



```
jacobi2D ( ) {
 for(i=1;i<1000;i++)
  for(j=1;j<1000;j++)
   a[i][j]=(b[i][j-1]+
            b[i][j+1]+
            b[i-1][j]+
          b[i+1][j])*c
}
```

**Original code**

```
jacobi2D_icp ( ) {
 start_inter_core_pref( )
 for(k=0;k<100;k++) {
  wait_and_swap(k+1)
   for(i=k*10;i<(k+1)*10;i++)
    for(j=1;j<1000;j++)
     a[i][j]=(b[i][j-1]+
              b[i][j+1]+
              b[i-1][j]+
            b[i+1][j])*c
 }
 end_inter_core_pref( )
}
```

**Main thread code**

```
jacobi2D_pslice ( ) {
 while(continue_prefetching){
  t = wait_and_swap(0)
  for(i=t*10;i<(t+1)*10;i++)
   for(j=1;j<1000;j+=8)
    load a[i][j], b[i][j]
 }
}
```

**Prefetch thread code**

Figure 4.2: Inter-core prefetching implementation for the 2D Jacobi kernel. Prefetch thread takes an argument that specifies the chunk it should prefetch. The calls to wait_and_swap() synchronizes the two threads before swapping processors.

cache while the prefetch thread moves on to start prefetching another chunk.

The tool uses standard Linux APIs – *setcontext*, *getcontext*, *swapcontext*, and pthreads to perform the swap. The library reduces the context switch overhead by a factor of 5. The entire thread management substrate requires only about 25 lines of code, not including comments. We use Linux's scheduler affinity interface to "pin" kernel threads to particular cores, while our thread management system moves logical threads between them.

Figure 4.2 shows the code for a 2D Jacobi implementation and the corresponding main thread and p-slice code. We use aligned chunking and each chunk is a sequence of the outer loop's iterations. To make the p-slice efficient, we remove all the arithmetic operations, and make it touch each cache line only once by incrementing $j$ by 8 in the inner loop since each cache line holds 8 elements. This transformation speeds up execution by a factor of 2.2× on a Core2Quad machine using four cores.

### 4.1.4 Impact of cache coherence

Cache coherence concerns do not have a significant impact for SMT based helper thread prefetching because both the helper thread and the main thread share all levels of cache. However, on inter-core prefetching, the underlying coherence protocol has a greater impact when chunks share data. This impacts both the generation of p-slices and the optimal chunk size. We will briefly discuss some of these issues here.

Write sharing is particularly problematic for inter-core prefetching. We do not allow helper threads to write, but if they prefetch data that the main thread writes to, it results in a slow upgrade transaction in the main thread and an invalidation in the helper thread rendering the prefetch useless. Read sharing has less of an impact, particularly when there exists a shared inclusive last level cache (like the L3 cache in Nehalem). In that case, both the main thread and the prefetch thread can get data from the shared cache. However, with an exclusive shared cache (L3 cache in Opteron), the prefetch thread will need to get the data from the other core (via cache-to-cache transfer) rather than from the shared cache. This is not a problem except that on this architecture cache-to-cache transfers are slow – they take 4 to 5 times as long as going to the L3 cache [HMN09]. The Core2Quad, with no global shared cache, has similar issues. Adding additional prefetch threads can potentially hide this extra latency and avoid any impact on the main thread.

The degree of sharing between chunks depends significantly on the chunk size. For instance, if neighboring loop iterations share data, a larger chunk size that covers many iterations will cause less inter-chunk sharing than a smaller chunk size. For the 2D Jacobi case in Figure 4.2, using one iteration per chunk means that the main thread and prefetching threads share 50% of read data, while using 10 iterations per chunk results in less than 10% shared data.

Table 4.1: The three processors have very different memory hierarchies that lead to different optimal points of operation for inter-core prefetching.

| **System Information** | **Intel Core2Quad**<br>**Intel Core2Quad** | **Intel Nehalem**<br>**Intel Nehalem** | **AMD Opteron**<br>**AMD Opteron** |
|---|---|---|---|
| CPU Model | Harpertown | Gainestown | Opteron 2427 |
| No of Sockets×<br>No of Dies×<br>No of cores | 2×2×2 | 1×1×4 | 1×1×6 |
| L1 Cache size | 32KB | 32KB | 64KB |
| L1 hit time | 3 cycles | 4 cycles | 3 cycles |
| L2 Cache size | 6MB (per die) | 256KB private | 512KB private |
| L2 hit time | 15 cycles | 10 cycles | 15 cycles |
| L3 Cache size | None | 8MB shared | 6MB shared |
| L3 hit time |  | 38 cycles | 36 cycles |
| Data TLB capacity (Level-1, Level-2) | 16, 256 | 64, 512 | 48, 512<br>48, 512 |
| Memory access latency | 300-350 cycles | 200-240 cycles | 230-260 cycles<br>230-260 cycles |
| Swapping cost | $2\mu s$ | $1.7\mu s$ | $1.7\mu s$ |

## 4.2 Methodology

This section describes the processors and applications we use to experimentally evaluate inter-core prefetching. The next section presents the results of those experiments.

### 4.2.1 Processors

Inter-core prefetching depends on the details of the processor's memory hierarchy and interconnect. Currently-available processors vary widely in the number of caches they provide on chip, the geometry of those caches, the interconnect between them, and the policies used to manage them. Table 4.1 summarizes the three processors we use to evaluate inter-core prefetching. On the Core2Quad, we use only one core per die, so that the L2 acts like a private cache.

All of the processors have hardware prefetching enabled. This allows us to evaluate whether inter-core prefetching effectively targets cache misses traditional

Table 4.2: Our benchmarks, with memory footprint, last level cache misses per 1000 instructions on Opteron, the number of loops where inter-core prefetching is applied, and the chunking technique used.

| **Benchmark name** | **Suite, Input** | **Memory footprint** | **LLC misses / 1000 inst** | **No of ICP loops** | **Chunking technique** |
|---|---|---|---|---|---|
| BT | NAS, B | 1200 MB | 19.96 | 20 | Aligned |
| CG | NAS, B | 399 MB | 20.86 | 5 | Aligned |
| LU | NAS, B | 173 MB | 17.42 | 14 | Aligned |
| MG | NAS, B | 437 MB | 11.74 | 6 | Unaligned |
| SP | NAS, B | 314 MB | 19.61 | 35 | Unaligned |
| Applu | Spec2000, Ref | 180 MB | 13.09 | 8 | Aligned |
| Equake | Spec2000, Ref | 49 MB | 28.53 | 4 | Aligned |
| Swim | Spec2000, Ref | 191 MB | 25.16 | 4 | Aligned |
| Lbm | Spec2006, Ref | 409 MB | 19.95 | 1 | Aligned |
| Libquantum | Spec2006, Ref | 64 MB | 24.45 | 6 | Aligned |
| Mcf | Spec2006, Ref | 1700 MB | 47.57 | 6 | Both |
| Milc | Spec2006, Ref | 679 MB | 27.34 | 20 | Aligned |
| Svm-rfe | Minebench | 61 MB | 16.97 | 1 | Aligned |

prefetchers cannot handle.

### 4.2.2 Applications

To evaluate inter-core prefetching, we use a collection of applications from Spec2000, Spec2006, the NAS benchmark suite, and MineBench [PLL+06]. Inter-core prefetching is only of interest for applications that are at least partially memory bound. For applications that are not memory bound, the technique will have little effect. To identify memory bound applications, we used performance counters to count last-level cache misses. If the application incurred more than 10 cache misses per thousand instructions on the Opteron, we included it in our test suite. Table 4.2 provides details about the workloads and inputs.

The table also lists how many loops we applied inter-core prefetching to and what type of chunks we used. For the NAS benchmarks we used the “W” inputs for profiling. For Spec2000 and Spec2006 we used the train inputs. Svm-rfe does

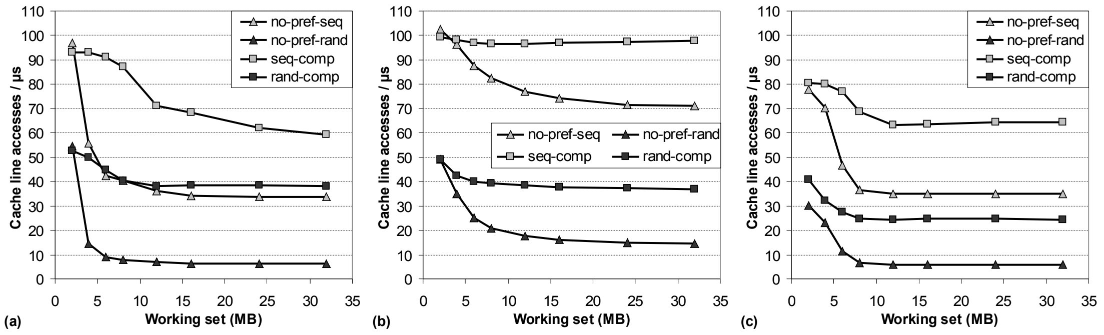


Figure 4.3: Inter-core prefetching throughput for – (a) Core2Quad, (b) Nehalem, (c) Opteron. As working set size increases to overflow one processor's cache capacity, inter-core prefetching prevents performance from dropping by using a cache on another processor to stage data.

not include multiple inputs, so we profiled on the same input we used to collect performance results.

## 4.3 Results

We evaluate inter-core prefetching in four stages. First, we use simple microbenchmarks to measure its potential and understand some of its tradeoffs. Then, we evaluate the technique's application-level impact using the workloads described in Section 4.2. Next, we evaluate inter-core prefetching's effect on power and energy consumption. Finally, we compare inter-core prefetching's performance across cores to a version of the technique that targets SMT contexts, and also with data spreading, another software-only cache optimization.

### 4.3.1 Microbenchmarks

To establish inter-core prefetching's potential, we use a simple microbenchmark that allows us to study the interplay between the chunk size, the number of prefetching threads, and the ratio of computation to memory access. Our microbenchmark accesses cache lines either sequentially (in virtual address space) or pseudo-randomly – in that case accessing each cache line in a region of memory exactly once, but in random order. The order is deterministic, though, allowing the prefetch thread to predict it accurately (this mimics dereferencing an array of pointers, for example). After accessing a cache line, the microbenchmark does a configurable number of arithmetic operations. We consider two cases: The *base* configuration executes four arithmetic operations per cache line access, and the *comp* configuration issues 32 operations. This gives four different microbenchmark configurations: rand-base, rand-comp, seq-base, and seq-comp.

### The potential of inter-core prefetching

We first use the microbenchmarks to establish the overall potential for inter-core prefetching as the working set size changes. Figure 4.3 shows the throughput (measured as cache lines accessed per $\mu$s) of seq-comp and rand-comp for the three machines. It compares performance with and without inter-core prefetching over a range of working set sizes. The chunk size is 256 KB, and it uses one prefetch thread (two cores total).

For the microbenchmark without prefetching, throughput is high when the working set fits in local cache, but drops off (sometimes precipitously) when it no longer fits. Inter-core prefetching significantly (and, in one case, completely) mitigates that effect. For working sets that fit within the processor's cache, inter-core prefetching adds a small amount of overhead, reducing performance by between 3 and 4%. As the working set size grows, however, the benefits of inter-core prefetching are potentially large. For instance, for sequential access, throughput improves 1.76×, 1.38× and 1.85× for Core2Quad, Nehalem, and Opteron, respectively. These gains are in addition to the performance that the hardware prefetcher provides. The gains for random accesses are even larger: 6×, 2.5× and 4×, respectively. This is primarily due to the fact that the hardware prefetcher is ineffective for these access patterns.

The performance gains for random accesses are particularly interesting, since inter-core prefetching delivers between 2.5 and 6× improvements in performance using just two threads. It may seem counterintuitive, since the number of cores only doubled relative to running without prefetching. This is an important result – inter-core prefetching is not bound by the limits of traditional parallelism (i.e., linear speedup). Instead, the only limit on speedup it offers is the ratio of memory stall time to compute time in the pipeline. When that ratio is large (i.e., the processor stalls for memory frequently), and the stalls can be removed by a small number of helper cores, we can achieve speedups well above linear. Therefore, inter-core prefetching is more than a fall-back technique to use when thread-level parallelism is not available. For some applications inter-core prefetching will be more effective than conventional parallelism. Of course, in those cases, a combination of traditional parallelism and inter-core prefetching is likely even better.

Looked at another way, inter-core prefetching achieves large speedups because it adds memory-level parallelism to applications that otherwise lack it. This is not obvious in the 2-core case, because we just move the data accesses from one core to the other. However, not only do the p-slices sometimes remove data dependences between loads that may have been in the code, they also minimize the number of instructions executed between loads and thus maximize the number of loads in the instruction window at one time.

Figure 4.4 demonstrates that inter-core prefetching moves almost all of the misses from the main thread to the prefetch thread. As long as the chunk fits in the L2 cache (for Opteron in this case), inter-core prefetching neither increases nor decreases L2 cache misses, it just performs them earlier and off the critical path. The speedup comes from the fact that we can access the data more quickly, and with more parallelism, than with a single thread.

In effect, inter-core prefetching partitions the work the application is doing into two, time-intensive parts: Executing non-memory instructions and waiting for cache misses. In this respect, it is similar to a decoupled access/execute [Smi82] approach to computation. If a single thread must handle both tasks, it performs poorly at both: Memory stalls inhibit instruction level parallelism, and dependent non-memory operations retard memory level parallelism by filling the instruction window. Allocating a core to each task removes these counterproductive interactions and increases the performance of both. Additionally, it separates prefetch

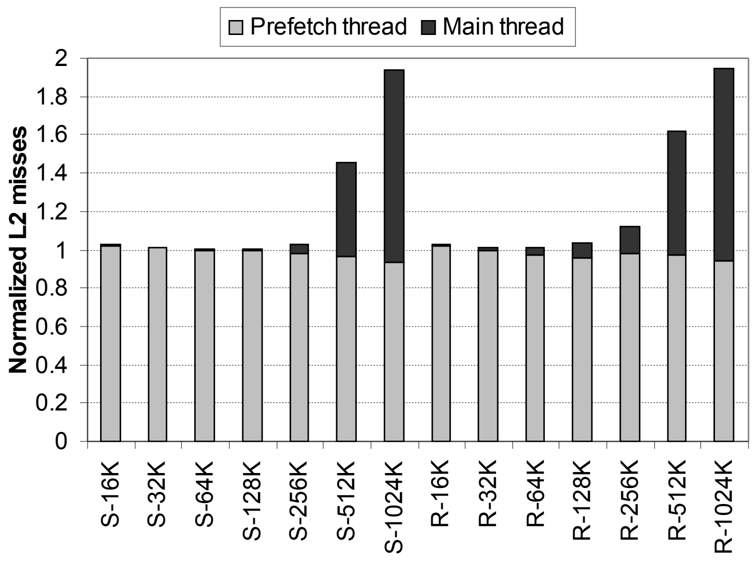


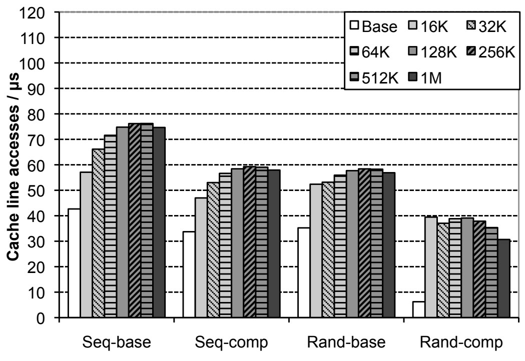
Figure 4.4: Misses incurred by the main thread and prefetch thread for different chunk sizes (R-16K means random access, 16 KB chunk), relative to the number of misses incurred in the baseline (no inter-core prefetching).

accesses and what demand misses the main thread still experiences onto separate cores, potentially increasing total memory bandwidth.

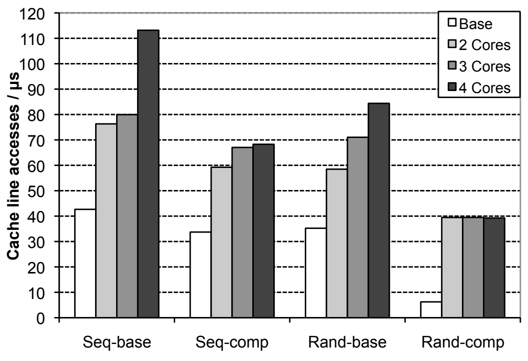
### Chunk size and the number of prefetch threads

Once the p-slice is generated, two parameters remain to be set – the chunk size and the number of prefetching threads (i.e., the number of cores to use). Changing the chunk size changes the level of the cache hierarchy prefetching will target and how completely we try to fill it, but it also determines how much swapping overhead occurs. For small chunk sizes, the frequent swaps might dominate the gains from prefetching.

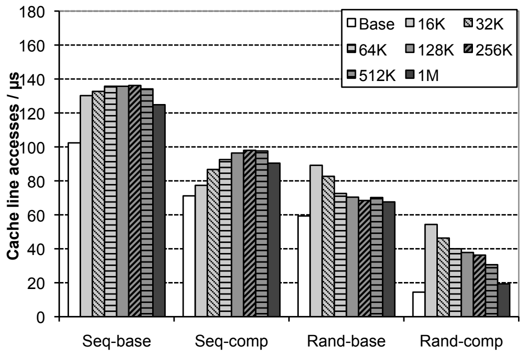
Increasing the number of prefetching threads can improve performance if the prefetching threads have difficulty keeping up with the main thread, since using additional cores to prefetch increases the amount of memory level parallelism the system can utilize. However, allocating more cores to prefetching increases power consumption and also means more cores are unavailable to run other threads.



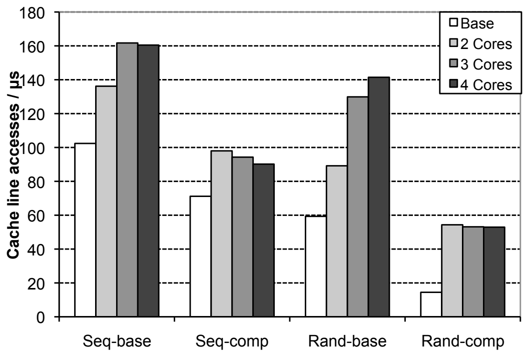
(a) Core2Quad, varying chunk size



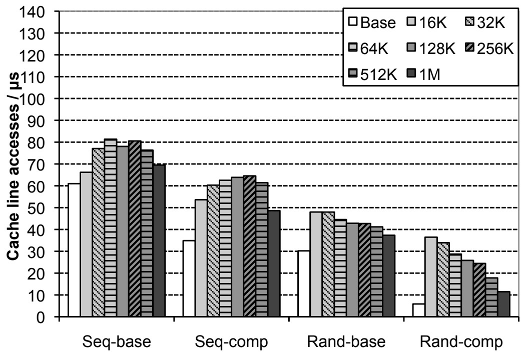
(b) Core2Quad, varying core count



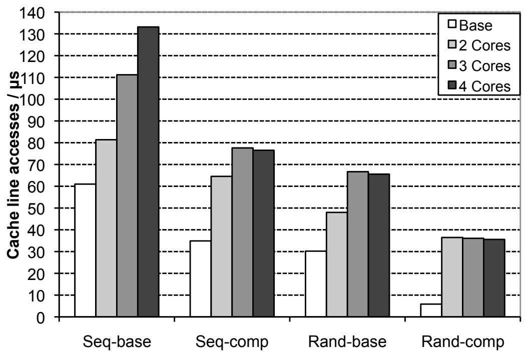
(c) Nehalem, varying chunk size



(d) Nehalem, varying core count



(e) Opteron, varying chunk size



(f) Opteron, varying core count

Figure 4.5: The graphs at left show the impact of chunk size on performance for our microbenchmark. The right hand figures measure the effect of varying the number of prefetching threads. Each pair of figures is for a different processor. Inter-core prefetching is especially effective for random access patterns that the hardware prefetcher cannot handle. For threads with less computation per memory access, more than one prefetching thread is necessary for optimal performance.

To understand the interplay of chunk size, compute intensity, and number of prefetch threads, we measure throughput for our microbenchmark with a 32 MB working set while varying the chunk size and the number of prefetch threads.

Figure 4.5 shows the results for our three processors. The figures on the left measure the impact of chunk size, while those on the right measure the effect of total core count. For the Core2Quad, a chunk size of 128 or 256 KB gives the best performance regardless of access pattern, but for other processors sequential accesses favor larger chunks between 64 and 256 KB, while 16 KB chunks perform best for random accesses.

The performance gains are again largest for the random access patterns, where there is limited benefit from hardware prefetching. For 16 KB chunks, inter-core prefetching improves Nehalem throughput by 3.75× and Opteron's performance by 6×.

The right hand graphs in Figure 4.5 show the impact of using multiple prefetching cores for the optimal chunk size from the left hand graphs. Adding additional prefetchers improves performance almost linearly up to four cores (three prefetchers) for seq-base on the Core2Quad and Opteron machines. Nehelam sees benefits with up to three cores. For seq-comp, conversely, one or two prefetchers give the best results. This is an expected result: When the main thread does little computation for each memory access, it is difficult for a single helper thread to execute as quickly as the main thread. For the random access patterns, results are similar.

It is not surprising that the optimal number of cores varies by memory access pattern and computation/memory ratio. However, the optimal chunk size also varies, especially by access pattern. This implies that the optimal chunk size depends not only on a particular cache size, but also at which level of the memory hierarchy the most misses are occurring, the effectiveness of the prefetcher, and the relative cost of migration/synchronization. For example, with random access, the program runs more slowly. This reduces the relative cost of migration, so smaller chunks are desirable. In fact, the migration overhead is amortized to the point that targeting the L1 cache is the best strategy for Nehalem and Opteron. For

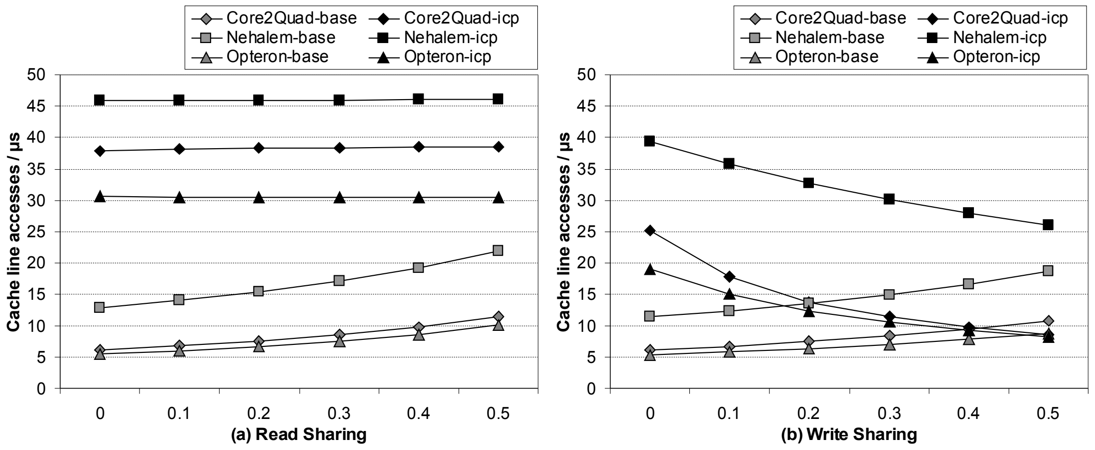
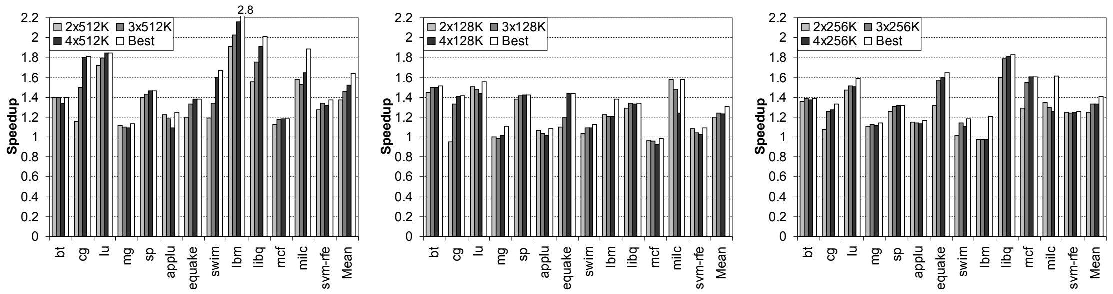


Figure 4.6: Performance improvement for different degree of read/write sharing using inter-core prefetching. Sharing factor of 0.3 means adjacent chunks share 30% of data.

sequential access (because it is harder to amortize the swap cost and because the hardware prefetcher is addressing the L1 cache fairly well), the sweet spot seems to be targeting the L2 cache. Less obvious is why the Core2Quad, with its huge L2 cache, tails off at 1 MB chunk size. However, inter-core prefetching allows us to not only pre-fill caches, but also TLB entries. The Core2Quad has 256 TLB entries per core, just enough to hold 1 MB if there are absolutely no conflicts in the TLB. Therefore, the optimal chunk size is just a bit smaller than the maximum space supported by the TLB.

We have also found that prefetching each chunk in reverse is helpful in some cases. By prefetching the same chunk data, but in the opposite order that the main thread will access it, inter-core prefetching can target multiple levels of the cache hierarchy at once – even if a chunk is too big for the L1 cache. In this case, the last data prefetched will be in the L1 when the main thread arrives and tries to access it. In general, though, this technique is more difficult to apply, so we did not include it in our experimental results shown.



(a) Core2Quad (b) Nehalem (c) Opteron

Figure 4.7: Inter-core prefetching provides good speedups (between 20 and 50% on average) without tuning the chunk size and thread count on a per-application basis. With that level of tuning, performance rises to between 31 and 63% (the "best" bars).

**Inter-chunk sharing**

To quantify the effect of inter-chunk data sharing, we modify our microbenchmark by adding a configurable sharing factor. Figure 4.6 shows the effect of inter-chunk sharing across different machines in the rand-comp case for both read and write sharing. We use 16 KB chunks with one prefetch thread and vary the fraction of shared data from 0 to 50%. As expected, without inter-core prefetching, throughput improves with the sharing because of the increase in locality. With inter-core prefetching, throughput stays the same for read sharing, but degrades rapidly for write sharing. Cache to cache transfer latency is the critical factor for write sharing. Nehalem, having the best cache to cache transfer latency among the three machines, is the most tolerant of write sharing and sees improvement even when the main thread and prefetcher share half their data. Core2Quad, with a private cache design, has expensive cache to cache transfer operations, so inter-core prefetching hurts performance when more than 40% of data is shared.

## 4.3.2 Application Performance

This section examines the effectiveness of inter-core prefetching on real applications, with the p-slices generated as described in Section 4.1. Figure 4.7 displays inter-core prefetching's impact on our benchmark suite. The first three

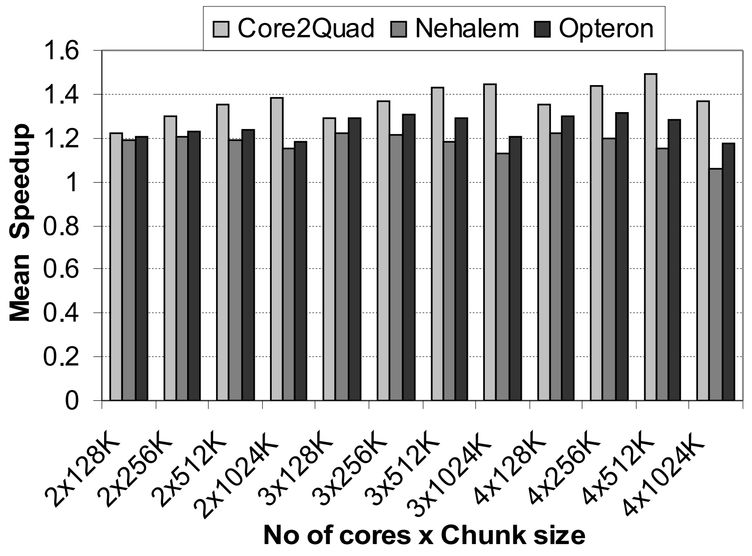


Figure 4.8: Mean speedup across all benchmarks for different combinations of chunk size and number of cores.

bars represent results for a single chunk size (that gave good performance across the entire benchmark suite) per architecture, varying the number of cores. The last bar gives the result that uses the best chunk size and core count per application (out of the 12 combinations– 2, 3, 4 cores and 128, 256, 512, 1024 KB chunks).

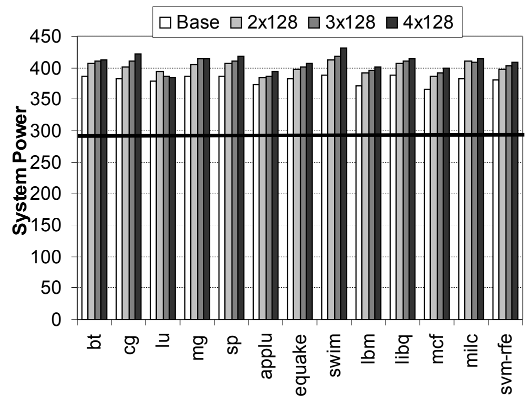
Overall, prefetching improves performance by between 20 and 50% on average without tuning the chunk size or core count for each benchmark. Per-benchmark tuning raises performance gains to between 31 and 63%. A further step would be to tune the parameters on a loop-by-loop basis.

Performance gains are largest for Core2Quad, because it has the largest cache miss penalty. For some applications, the gains are far above the average: Prefetching speeds up LBM by 2.8× (3 cores and 1024 KB chunks) on the Core2Quad and MILC sees improvements of nearly 1.6× for Nehalem. Even a complex integer benchmark like MCF gets 1.6× on Opteron. MCF also shows the impact of microarchitectural differences across machines on performance improvements. Unlike Opteron, Nehalem does not see any improvements and Core2Quad sees moderate speedup of 20%. This variation is due to the difference in cache hierarchy, window size, hardware prefetcher implementation, coherence protocol, etc. across these machines. To understand these differences, we measure the percentage of execution time covered by loops selected for inter-core prefetching (measured in the baseline case). Nehalem and Core2Quad spend between 44 and 47% of execution in those loops. Opteron spends 64%, which accounts for the larger overall impact on performance.

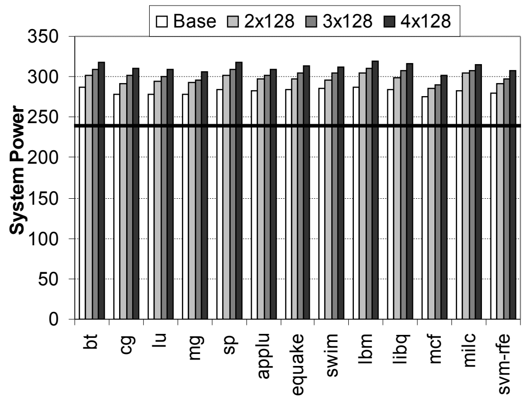
The effectiveness of increased prefetching threads varies widely among applications. For some (e.g., CG) additional prefetching threads are useful on all three architectures. For CG running on Nehalem with one prefetch thread, the main thread has to wait for the prefetch thread 89% of the time. With two and three prefetch threads, that number goes down to 25% and 0.4%, respectively. We see similar data for CG on Core2Quad and Opteron. For others (e.g., applu), adding threads hurts performance. If thread count has a strong impact on performance, the trend tends to be consistent across all three processors. The optimal number of cores is a function of the ratio of memory stalls to computation, which tends not to change across architectures.

Figure 4.8 shows results averaged across all benchmarks, for a wider range of chunk sizes for the three machines. We see again from these results that speedups are somewhat tolerant of chunk size, although there are some poorer choices to be avoided.

Looking closely, ICP exploits the latency gap between different levels of memory and becomes the most useful when these gaps are exposed due to the lack of memory level parallelism. In Core2Quad, the latency ratio between accessing the DRAM and last level cache (L2 cache) is around 20, and we get 63% speedup on average. Compared to that, the ratio is around 6 for Nehalem, and the performance improvement is 31% on average. For Nehalem, there is another latency gap between L2 and L3 cache that also contributes. We see similar behavior for the Opteron system. This gives us an intuition about how ICP may perform in a future system where there can be a different cache organization and different access latencies. The latency gap between the last level cache and DRAM has the strongest impact. But if we prefetch in a cache other than the last level cache, the benefit is more,



(a) Core2Quad power



(b) Nehalem power

Figure 4.9: Inter-core prefetching increases power consumption slightly compared to the non-prefetching version of the application, but the prefetching threads consume much less power than the main thread. The horizontal line denotes idle power on each system.

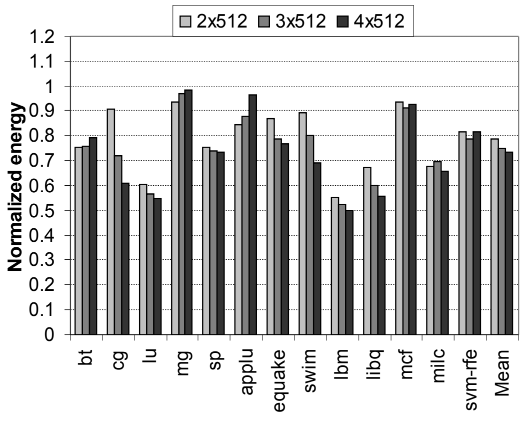
and it depends on the latency gap between the corresponding levels of the cache hierarchy.

### 4.3.3 Energy Considerations

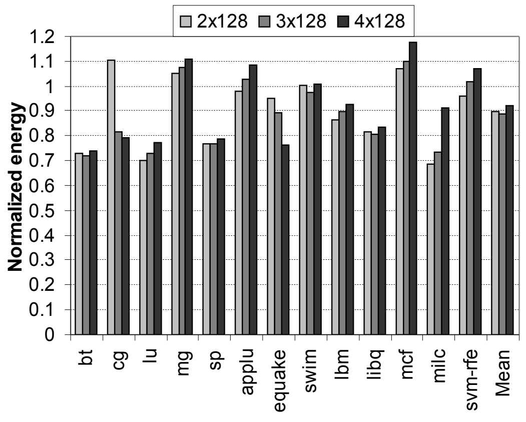
Inter-core prefetching has two competing effects on application energy consumption. The technique increases performance, which would reduce energy consumption if power remained constant. However, using multiple cores for execution increases power consumption both because multiple cores are active and because the throughput of the main thread increases.

To measure the total impact on energy, we measure total system power and energy with a power meter. The measurement includes everything in the system, including the power supply and its inefficiency, so the results are not directly comparable with simulation studies that focus only on processor power.

Figures 4.9 and 4.10 show results for Nehalem and Core2Quad. We were unable to measure results for the Opteron because of system administration issues. Our measurements show that the applications running without prefetching require 282 W (Nehalem) and 381 W (Core2Quad) on average (i.e., total system power



(a) Core2Quad energy



(b) Nehalem energy

Figure 4.10: Inter-core prefetching's performance gains counteract the increase in power consumption it causes, resulting in a net reduction in energy of between 11 and 26%. Measurements in the graph are normalized to the application without inter-core prefetching.

increased by 42 W and 87 W while running the application compared to an idle system). Inter-core prefetching with a single prefetcher increases power consumption by 14 W (Nehalem) and 19 W (Core2Quad), and adding another prefetching thread requires an additional 6 W.

In terms of energy, the performance gains that inter-core prefetching delivers more than compensates for the increased power consumption. Per-application energy drops by 11 and 26% on average for the two architectures, and as much as 50% for some applications.

### 4.3.4 Comparison with SMT Prefetching

Previous work on helper-thread prefetching focused on SMT machines. This section compares that approach with inter-core prefetching.

To provide a reasonable comparison, we use the same chunk and p-slice generation techniques that we use for inter-core prefetching, but we run the p-slice on the second context of one of Nehalem's SMT cores. Since the main thread and prefetching thread will run concurrently on the same core, there is no need for context switches or the associated overhead.

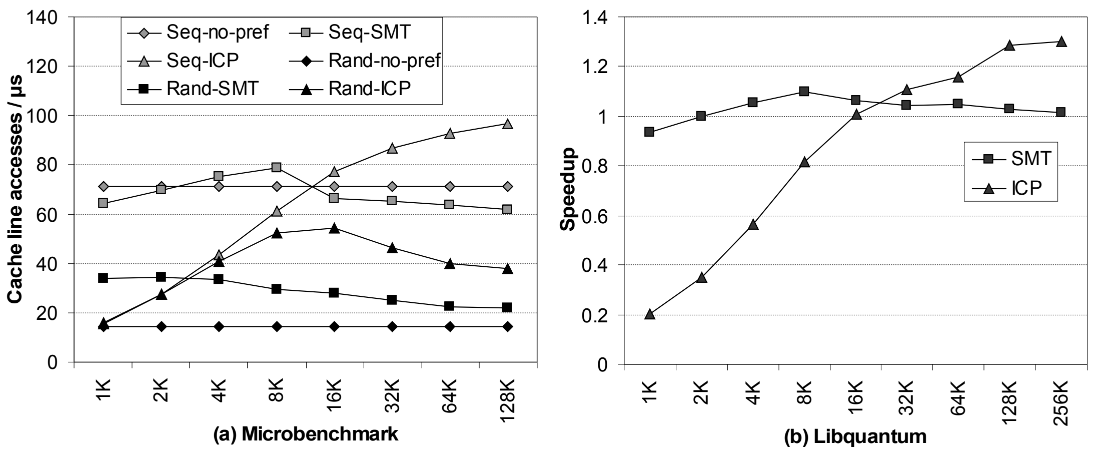


Figure 4.11: Comparing inter-core prefetching and SMT prefetching for different chunk sizes in Nehalem. On SMT, smaller chunk sizes give better performance because migration is not necessary and the prefetcher can target the L1 cache. However, using multiple cores still results in better overall performance.

Figure 4.11(a) compares the performance of applying prefetching across SMT contexts and applying it across cores for Nehalem using our microbenchmark. We see from these results that SMT prefetching favors smaller chunk sizes since they minimize interference between the helper thread and the main thread in the L1 cache. With very small chunks, of course, CMP prefetching is not effective because of the cost of swapping. However, with large chunks inter-core prefetching easily outperforms the best SMT result. Figure 4.11(b) shows the same comparison for one of our Spec2006 benchmark, *libquantum*. This application has regular memory access pattern, and so it follows the trend of sequential access in Figure 4.11(a).

Inter-core prefetching, even on architectures where SMT threads are available, benefits from the absence of contention for instruction execution bandwidth, private cache space, TLB entries, cache bandwidth, etc. This lack of contention allows the prefetcher threads to run very far ahead of the main thread and makes it easier to fully cover the latency of cache misses. As a result, inter-core prefetching is vulnerable to neither useless late prefetches nor early prefetches whose data are evicted before the main thread accesses it. This also explains why the hardware prefetcher cannot completely eliminate the need for inter-core prefetching in case of the fully predictable (e.g. sequential) access pattern. In addition, the hardware prefetcher will not cross page boundaries, while inter-core prefetching ignores them.

### 4.3.5 Comparison to data spreading

Inter-core prefetching shares some traits with our previous work on data spreading (Chapter 3), which also uses migration to eliminate cache misses. Under that technique a single thread periodically migrates between cores, spreading its working set across several caches. For applications with regular access patterns and working sets that fit in the private caches, data spreading converts main memory accesses into local cache hits. For more irregular access patterns, it converts them into cache-to-cache transfers. The results show that it can improve performance by up to 70%.

Inter-core prefetching and data spreading are complimentary, but they differ in several ways. First, data spreading is only useful when the working set of the application fits within the aggregated caches. Inter-core prefetching works with working sets of any size. Second, data spreading uses a single thread, and, as a result, does not incur any power overheads: only one core is actively executing at any time. Inter-core prefetching actively utilizes multiple cores. Finally, as datasets grow, data spreading requires more and more cores. In many cases, inter-core prefetching gets full performance with 2 cores, even with very large data sets.

Figure 4.12 compares inter-core prefetching and data spreading on Jacobi. The figure measures speedup relative to a conventional implementation on the same architecture. The data show that data spreading provides better performance when Jacobi's data set fits within the aggregate cache capacity of the system. Inter-core prefetching offers better performance for larger data sets. The technique also does not require larger private caches. This helps to effectively apply inter-core prefetching in state of the art multicores with moderately sized private L2 caches.

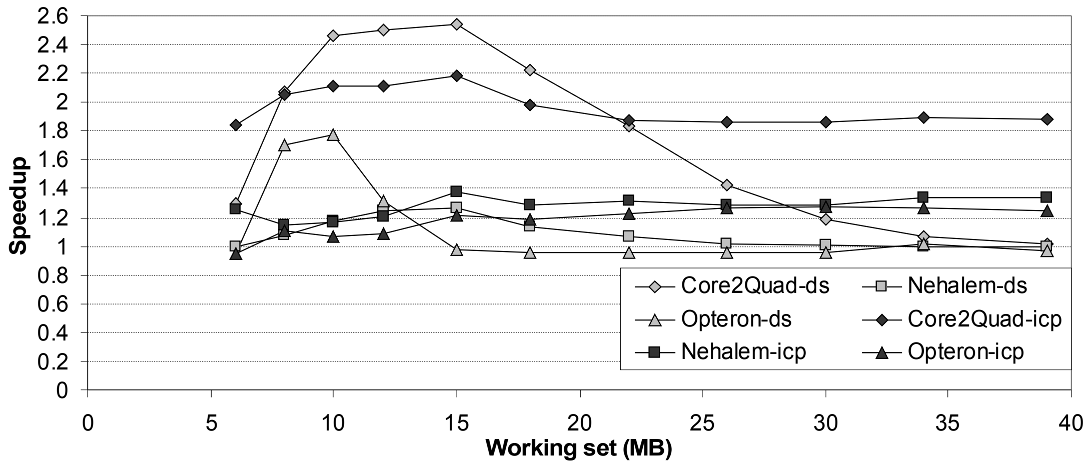


Figure 4.12: Inter-core prefetching and data spreading are complimentary techniques. Data spreading delivers better performance and power savings for small working sets, but for larger working sets, inter-core prefetching is superior. Speedups are relative to a conventional implementation.

## 4.4 Conclusion

This chapter describes inter-core prefetching, a technique that allows multiple cores to cooperatively execute a single thread. Inter-core prefetching distills key portions of the original program into prefetching threads that run concurrently with the main thread but on a different core. Via lightweight migration, the main thread follows the prefetching threads from core to core and finds prefetched data waiting for it in its caches. Our results show that inter-core prefetching can speed up applications by between 31 and 63%, depending on the architecture, and that it works across several existing multi-core architectures. Our results also show that, although it uses multiple cores, it can reduce energy consumption by up to 50%. Inter-core prefetching demonstrates that it is possible to effectively apply helper thread-based techniques developed for multi-threaded processors to multi-core processors, and even overcome several limitations of multithreaded prefetchers.

## Acknowledgments

Portions of this research were funded by the US National Science Foundation grants CCF-0702349 and NSF-0643880.

This chapter contains material from “Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads”, by Md Kamruzzaman, Steven Swanson and Dean M. Tullsen, which appears in ASPLOS’11: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. The dissertation author was the primary investigator and author of this paper.

# Chapter 5

# Coalition Threading: Adapting Non-Traditional Parallelism for Parallel Applications

Previous chapters demonstrate the potential of inter-core prefetching and software data spreading in the absence of traditional parallelism (i.e., serial code). These non-traditional parallelism (NTP) techniques provide parallel speedup (several cores or hardware contexts execute a program faster than a single core) without changing the number of logical threads (e.g., pthread or MPI threads) running the computation. This chapter shows that NTP is also effective even when traditional parallelism exists. For many loops, the best parallel solution is to apply non-traditional parallelism on top of traditional parallelism rather than using only traditional parallelism. We can exploit this by using a combination of these two types of parallelism that applies different techniques to different parts of an application to increase scalability. We call this combination *coalition threading*, and in this chapter we examine, in particular, the combination of traditional parallelism expressed as pthreads and OpenMP constructs, paired with inter-core prefetching (ICP) described in Chapter 4. ICP uses a helper thread to prefetch data into a local cache on one core, then migrates execution to that core. This technique requires no hardware support and works across cores, sockets, and SMT contexts, so it is applicable to a wide range of current and future parallel microprocessors.

This chapter quantifies the benefits of coalition threading on a state-of-the-art multiprocessor, running a range of benchmarks, and describes compiler heuristics that can identify the loops in each benchmark most suited to non-traditional techniques. While coalition threading represents an opportunity for the programmer or compiler to increase parallel speedup and scalability, deciding when to apply each technique is not easy, and making a poor decision can adversely impact performance.

Our results show that coalition threading with ICP can outperform traditional threading in 39% of the 108 loops in eleven applications we studied, chosen to only include those that already saw some benefit from traditional parallelism. By selecting the best threading strategy on a loop-by-loop basis, we demonstrate that coalition threading improves total performance by 17% on average and up to 51% for some applications.

Achieving that level of performance requires that the compiler, programmer, or runtime system be able to determine when to apply each type of parallelism. We find that a standard machine learning technique, linear classification [FCH$^+$08], accurately classifies 98% of the loops where the decision is critical and 87% overall. Considering application performance, the heuristic achieves over 99.4% of the performance that an oracle could deliver.

This chapter addresses several contributions. It is the first investigation into the effectiveness of combining non-traditional parallelization techniques with traditional parallelization. It demonstrates that a combined approach can as much as double the performance of traditional parallelization alone for particular loops. We show that performance can be highly sensitive to the decision made for each loop. To handle that, this chapter develops heuristics that a compiler can easily implement to identify which combination of traditional and non-traditional parallelism will work best.

The remainder of this chapter is organized as follows. Section 5.1 describes coalition threading. Section 5.2 describes our methodology. Section 5.3 compares the performance of parallelization techniques. Section 5.4 derives and evaluates heuristics to identify the best parallel approach, and Section 5.5 concludes.

## 5.1 Coalition Threading

Coalition threading augments conventional parallel threads with helper threads to improve performance and scalability. Coalition threading works because many loops (virtually all, if you scale far enough) have limited scalability – non-traditional parallelism can pick up when traditional tails off. In addition, NTP is not bound by the limits of traditional parallelism. Both data spreading, and inter-core prefetching have demonstrated greater than $n$-fold speedups with $n$ helper threads. This means that NTP can be a profitable choice even for scalable loops.

In principle, coalition threading will work with many different kinds of helper threads, but we focus on inter-core prefetching, because it works on any cache coherent system and subsumes other techniques including SMT helper thread prefetching [CWT+01], shared cache prefetching [LDH+05], and also software data spreading.

The following section identifies the key issues that arise in coalition threading in general and with ICP in particular. Then we describe the implementation of our coalition threading framework.

### 5.1.1 ICP based coalition threading

Our coalition threading implementation uses ICP as the non-traditional component. For clarity, we use the term *par-ICP* to describe ICP implemented on top of traditional parallelism, and applied to each of the original parallel threads. *Seq-ICP* is ICP applied alone, running on top of a single-thread version of an application. Coalition threading refers to choosing the best technique (seq-ICP, par-ICP, or traditional parallelism) for a particular loop or for all the loops in an application.

Par-ICP applies inter-core prefetching to each original or *main* thread. If an application has $N$ main threads and ICP provides $M$ helper threads per main thread we need a total of $T = (N + N \times M)$ hardware contexts. When $N$ is one, we only exploit helper thread parallelism. Likewise when $M$ is zero, the application is running with just traditional parallelism.

Note that the choice for the value of $M$ is critical for parallel code. In prior work, it was assumed that other cores were otherwise idle, and adding more helper threads was close to free. For parallel code, there is typically a high opportunity cost, because even increasing $M$ by one reduces the number of main threads by a factor of $(2 + M)/(1 + M)$. For example, for a total of 24 threads, we can either have 24 main threads with no helper threads, 12 main threads each with one helper thread, or 8 main threads each with two helper threads. Recall that for ICP, more than one helper thread is only useful when it takes more time to prefetch a chunk than to execute that chunk. Thus, we want our prefetch threads to be as efficient as possible, and to use no more threads than necessary to leave more threads/cores for traditional parallelism. Fortunately, in most cases, ICP achieves nearly all its benefits with M=1.

Several new factors impact the effectiveness of ICP when applied to parallel code, including parallel data sharing and synchronization overhead.

Data sharing (including false sharing) has a stronger negative impact on the effectiveness of par-ICP than it has on seq-ICP. For read sharing, if the main threads share data then so do helper threads, both with each other and with other main threads. Thus, a single cache line can be in many caches. This uses the caches inefficiently and potentially leads to expensive upgrade events when a main thread eventually writes the line. Write sharing is even more critical, because more than one thread can now invalidate the prefetched data a helper thread has accumulated.

Conversely, if access to that shared data is protected by barriers or locks, ICP can be extremely effective. There are several reasons for this. First, synchronization overhead slows down the main thread, making it easier for prefetch threads to keep up. Second, those applications tend to scale poorly, so even if ICP is not helpful at low thread counts, it can be better than traditional parallelism at high thread counts. Third, loops with frequent communication are often vulnerable to load imbalance – in those cases it is often better to make every main thread faster rather than increase the number of main threads.

### 5.1.2 Our coalition threading framework

We develop a custom-built source to source translator based on the Rose compiler infrastructure [QSPK01] to implement coalition threading. This framework is flexible enough to apply other non-traditional techniques like shared cache prefetching [LDH$^+$05] or software data spreading (Chapter 3). However, ICP generally outperforms these techniques and so we do not demonstrate them in this chapter. The framework supports both OpenMP or pthread code. In the case of OpenMP applications, the framework first converts the OpenMP pragmas into pthread constructs.

Our system decides on a loop-by-loop basis whether and to what extent to apply traditional threading or par-ICP. The framework uses a heuristic to decide the threading technique for each loop and automatically produces code that implements the selected threading technique. The new code generates a *thread group* of $(1 + M)$ threads for each of the original threads.

Threads within a thread group can trade places with one another, but cannot migrate across thread groups. We use the user-space thread migration system used for inter-core prefetching to make migration as cheap as possible.

The framework uses all $(N + N \times M)$ threads for parallel execution for loops selected for traditional threading. For par-ICP loops, the framework implements ICP within each thread group with one main thread and $M$ helper threads. The framework generates all the code necessary for splitting the loop into chunks and the coordination between main threads and helper threads required for swapping cores within a group. Each main thread provides necessary information (live-ins, chunk id) to the corresponding helper threads to prefetch the right chunk. However, the framework does not automatically create the code that will prefetch data. Instead the framework looks for a callback function associated with a loop that will act as the prefetch function. Generating the callbacks automatically is possible (e.g., as in [KY02]), but for the applications we study, we construct them by hand.

To maximize prefetching efficiency we introduce two optimizations over the original ICP implementation described in previous chapter. First, we use software

Table 5.1: Memory hierarchy information and migration cost for the AMD Opteron system used in our experiments.

| **CPU components** | **Parameters** |
|---|---|
| CPU model | AMD Opteron 6136 2.4 GHz |
| Number of cores | Quad-socket 32 cores |
| Level-1 (L1) cache | 64-Kbyte, 3 cycles |
| Level-2 (L2) cache | 512-Kbyte, 15 cycles |
| Level-3 (L3) cache | 6-Mbyte, 50 cycles |
| Cache to cache transfer latency | 120-400 cycles |
| Memory | 60-Gbyte DRAM, 150 cycles |
| Migration cost | $2 - 5\mu s$ |

prefetching instructions instead of regular loads to avoid accidental page faults. Second, we prefetch the chunk in reverse order – iterations that are to be executed last by the main thread are prefetched first by the helper thread. This ensures that the data the main thread will use first will be in the L1 cache even if our prefetch code and chunk size target the L2 cache.

## 5.2 Methodology

This section describes our experimental methodology – the benchmarks that we target and the systems that we use to evaluate coalition threading.

### 5.2.1 Evaluation Systems

We use a quad socket, 32-core state of the art AMD Opteron system running Linux 2.6. Table 5.1 gives more information about the system. We compile all codes using gcc4.1.2 with -O3 optimization level and keep hardware prefetching enabled for all experiments. To measure microarchitectural characteristics like L2 cache misses, we use hardware performance counters.

Table 5.2: List of the pre-parallelized benchmarks used for the evaluation of coalition threading. The table also shows the number of loops analyzed.

| **Benchmark Name** | **Suite Input used** | **Loops Evaluated** |
|---|---|---|
| BT | NAS, B | 23 |
| CG | NAS, B | 2 |
| LU | NAS, B | 14 |
| MG | NAS, C | 6 |
| SP | NAS, B | 35 |
| Ammp | SpecOMP, Ref | 6 |
| Art | SpecOMP, Ref | 2 |
| Equake | SpecOMP, Ref | 5 |
| Fluidanimate | Parsec, Native | 6 |
| Streamcluster | Parsec, Native | 2 |
| Graph500 | Graph500, 25–5 | 7 |

### 5.2.2 Applications

To evaluate coalition threading, we use memory intensive multi-threaded applications from the NAS parallel benchmark suite [BBDS93], SpecOMP benchmark suite [ADE$^+$01], Parsec [BKSL08], and Graph500 [RCMA10] workloads. Table 5.2 summarizes the particular applications and input sizes we use. We did not include SpecOMP benchmarks that were not written in C or C++ code. In addition, we had to exclude a few more benchmarks because Rose could not handle them.

In all, we target 108 loops (Table 5.2) from 11 applications. All of these 108 loops are programmer-parallelized and they include all the loops that contribute more than 0.5% of the total execution time of the corresponding benchmark when the benchmark executes with one thread.

## 5.3 Effectiveness of the Different Parallelization Techniques

This section measures the effectiveness of coalition threading on a variety of parallel workloads. All the data presented in this section apply for our large 32-core Opteron system.

### 5.3.1 Benefits of coalition threading

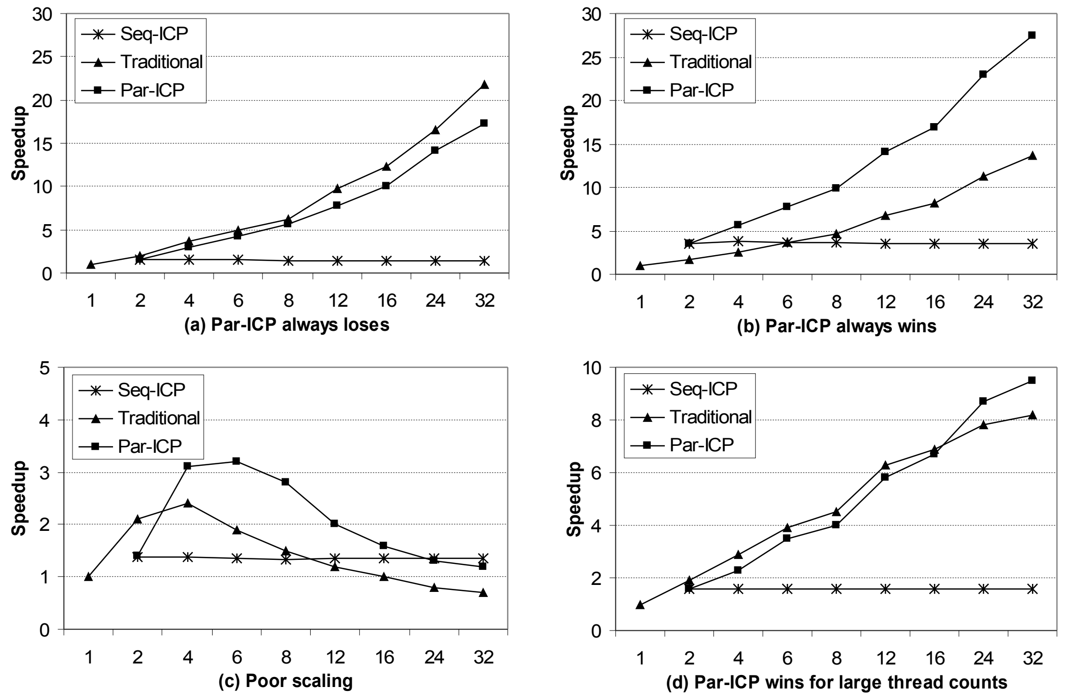


Figure 5.1: Loops show four different behaviors when we apply seq-ICP, traditional threading, and par-ICP for different thread counts (shown on the x-axis). Most applications contain loops from more than one group. Here, all four loops are from the benchmark *SP*.

The usefulness of coalition threading varies on a loop-by-loop basis, so coalition threading for a particular application may include traditional threading for some loops and a combination for others. We will use the term *traditional* when we use only the parallelism as indicated in the parallel programs themselves, seq-ICP means we only apply inter-core prefetching, par-ICP means we combine ICP with the existing parallelism, and coalition threading (CT) means we make an intelligent decision about which type of parallelism to apply to each loop.

We evaluate these techniques on each of the 108 loops in our applications, using between 1 and 32 total threads (and cores). All comparisons throughout the chapter apply for the same total number of threads (cores). So, the data point for 32 threads means either 32 main threads (traditional), or 16 main threads with one helper thread (par-ICP), or 1 main thread with 31 helper threads (seq-ICP). We consider other cases of par-ICP (more than one helper thread) in Section 5.3.2.

We see four different behaviors across the loops. Figure 5.1 compares the performance for four loops that represent these four classes of behaviors. The first class (a) are loops for which traditional threading performs better regardless of thread counts. For loops in class (b), par-ICP consistently outperforms traditional threading. The third class, (c), contains loops that scale poorly (or negatively) even for small thread counts with traditional threading. Behavior for the fourth class, (d), changes as the number of threads increases, despite the fact that these loops tend to have good scalability. In particular, par-ICP loses for small thread counts, but wins for large thread counts. All of these four loops are from one application, *SP*, demonstrating the importance of applying coalition threading on a per-loop basis.

Since all our benchmarks are parallel to start with, ICP alone (seq-ICP) is never the best choice for large core counts. In the more general case, seq-ICP would be an attractive case for many loops that lack parallelism. In our experiments, seq-ICP speeds up almost all of our loops (90 out of 108) over sequential execution, but never over parallel execution; thus we do not consider seq-ICP as an option for the compiler in this chapter.

We introduce the term $\textit{ICP-gain}_T$ for a loop to compare performance between traditional parallelization and par-ICP. It is the ratio of the maximum speedup using par-ICP and the maximum speedup using traditional threading while using no more than $T$ threads in total. So, for 32 threads, if a loop gets maximum speedup of 16$\times$ using traditional threading and maximum speedup of 20$\times$ using par-ICP, the loop has an $\textit{ICP-gain}_{32}$ of 1.25. In the case of coalition

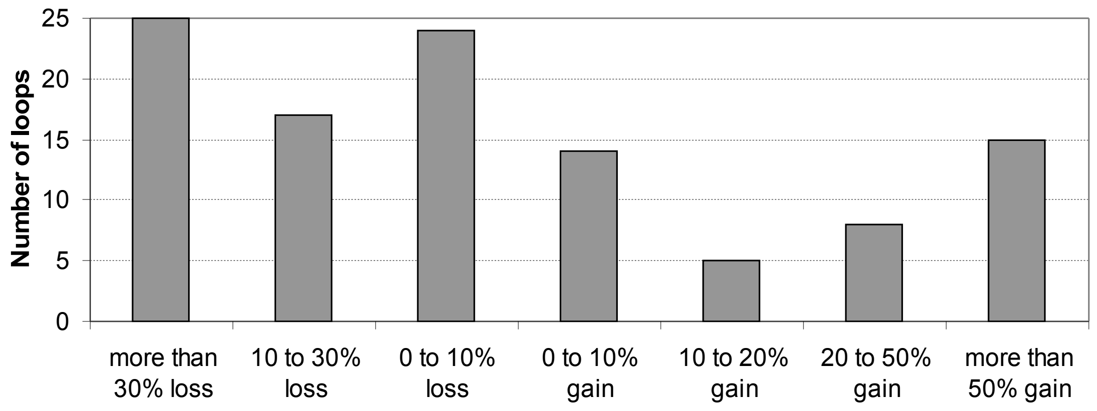


Figure 5.2: Histogram of 108 loops using *ICP-gain*$_{32}$ for the Opteron system. There are 15 loops that get more than 50% speedup.

threading, we seek to apply par-ICP for only those loops that have *ICP-gain*$_T > 1$.

Figure 5.2 shows the histogram of the 108 loops to represent the impact of applying par-ICP on each loop in the Opteron system. The x-axis shows different intervals of *ICP-gain*$_{32}$, and the y-axis shows the number of loops in that category. Applying par-ICP outperforms traditional threading in 42 loops in Opteron. There are 15 loops that gain at least 50% or more, and 7 loops see speedup as high as 2$\times$ or more on top of traditional parallelization.

There are a large number (38 out of 108) of loops where the difference between par-ICP and traditional is within $\pm$10%. The most common trend we see is for the par-ICP gains to increase with core counts, in fact we observe quite a few cases where the curves appear to be reaching a crossover point just at the limit of cores. Thus, as we continue to scale the number of cores, we expect the number of applications for which coalition threading is profitable to also increase.

Next, we try to understand the factors that cause some of the loops to scale better using par-ICP than traditional threading.

### 5.3.2 Understanding loop behavior

We analyze all 108 loops to identify the most common factors that cause the diverse behavior when we evaluate both traditional threading and par-ICP on each loop.

There are some loops where traditional threading always outperforms coalition threading, as shown in Figure 5.1(a). For these loops, adding more main threads provide better performance than using ICP, even if ICP has something to offer. The 59 loops in this class achieve 19% speedup on average when we apply ICP in the single-threaded case with one helper thread, but if we use that thread for parallel execution, we get, on average, an 83% speedup. These are highly parallel loops with little or no synchronization overhead, few coherence misses, and high instruction level parallelism (ILP) that hides the effect of memory latency, if any.

Conversely, Figure 5.1(b) represents the set of loops where par-ICP always outperforms traditional threading. This includes both loops that scale well and those that do not. The common factor is that ICP is highly effective in all of these cases. For the 16 loops in this category, executing in parallel gives 1.2$\times$ speedup on average for two threads ($N = 2, M = 0$), while ICP gives an average speedup of 1.9$\times$ using just one main thread and one helper thread ($N = 1, M = 1$). For some loops, we even see speedup as high as 3.5$\times$. These loops have a significant number of L2 misses per instruction, and have little memory level parallelism.

In our analysis, we also find several loops that scale poorly for traditional threading (overlaps with the prior category in some cases). These loops stop scaling beyond some thread count or even start scaling negatively, as in Figure 5.1(c). We observe three main causes for such behavior. First, some loops have limited scalability because there are not enough iterations to distribute over all threads. This creates load imbalance and does not amortize the cost of parallelism with only one iteration per thread. Second, there are some loops where computation increases significantly (linearly in some cases) with the number of threads. For example, threads can have a private copy of the shared data and can update that copy independently and merge afterward instead of using locks to access that shared data. Finally, some loops execute barriers very frequently and so synchronization time dominates the execution time as the loop is scaled. In all these three cases, par-ICP always works better than traditional parallelism for higher thread counts.

Some loops (total 33) exhibit more complex behavior. Their behavior changes more slowly with the thread counts, as in Figure 5.1(d). For small thread counts, performance for traditional and par-ICP is similar or par-ICP performs worse. But for larger thread counts, par-ICP starts outperforming traditional threading and the performance gap grows. This is often just a sub case of the previous category, but with lower synchronization overheads that are just starting to be exposed with high thread counts. We see just a couple of counterexamples (2 loops from $LU$ benchmark), where the opposite happens – par-ICP wins for smaller thread counts but loses for larger thread counts. These two loops show super linear parallel scalability when enough cores are used to fit the working set in the L2 caches, making prefetching less effective at those core counts.

### Multiple helper threads

We also measure performance using more than one helper thread per main thread. Multiple helper threads can be effective when applied to the single-threaded case. In our experiments, the speedup (averaged over all 108 loops) improves to 1.49× from 1.38× when we add one more helper thread ($N = 1, M = 2$ instead of $N = 1, M = 1$). However, in most cases, this improvement cannot compensate the reduced parallelism because of the decrease in the number of main threads. Additionally, more helper threads saturate off-chip bandwidth in some cases and we do not see the same degree of speedup as we scale to higher thread counts.

We found three loops where multiple helper threads provided better performance than using a single helper thread for the same total number of threads (e.g., $N = 8, M = 2$ works better than $N = 16, M = 1$). These loops each had poor scalability as in Figure 5.1(c). Using more threads for ICP reduced the number of main threads and improved performance. Even in these cases, the gains were small. So, we restrict ourselves to one helper thread per main thread in the remainder of the chapter. This also has the benefit of significantly reducing the decision space for the compiler.

Table 5.3: List of the 12 loop characteristics that we measure for developing the linear classification based heuristic.

| **Name** | **Description** |
|---|---|
| PO | Parallelization overhead |
| CDR | Chunk data reuse |
| L2MI | L2 misses per instruction |
| TLBL2IN | L2 misses for TLB walking per instruction |
| DTLBIN | TLB misses per instruction |
| SSE | No of arithmetic operations per instruction |
| BRANCH | No of branches per instruction |
| IPC | Instructions per cycle |
| DRAM | DRAM accesses per instruction |
| FETCH | Per instruction stalls for not fetching |
| DISP | Per instruction stalls for not dispatching |
| LSQS | Per instruction stalls for load store queue full |

## 5.4 Heuristics to apply coalition threading

Coalition threading is the process of making a decision about when to apply what type of parallelism to each loop. Because we filter our benchmarks for existing parallelism, we focus here on deciding between traditional parallelism and par-ICP. For the more general case, seq-ICP must also be considered.

Predicting when par-ICP is profitable for a loop, then, is essential. Our compiler tool chain profiles (single-threaded) and analyzes each loop to determine how to extract the most performance using the available CPUs. Depending on the loop's characteristics, it will either apply par-ICP or traditional threading. This section describes the heuristics that our compilation framework uses.

We use a standard classification algorithm in machine learning, linear classification [FCH+08], to develop our heuristics. It exploits profile information that we capture from the single-thread execution of the application to classify loops. We describe this next.

### 5.4.1 Linear Classification

A linear classifer [FCH+08] is a simple and robust binary classifier widely used in machine learning. It uses a hyperplane of n-1 dimension to best separate a set of n-dimensional points to two classes. The linear classifier library like *liblinear* [FCH+08] automatically extracts a weight vector from the input training set to represent the hyperplane. For a particular test point, the classifier computes the dot product of the weight vector and the feature vector of the test point and makes a decision based on that. The number of points that the hyperplane can accurately classify to the correct class determines the accuracy of the classifier.

In our case, we consider each loop as an n-dimensional point where $n$ is the number of loop characteristics or features that we measure. There are 12 such characteristics, shown in Table 5.3. These characteristics capture two main things – factors that impact the scalability of traditional parallelism and factors that impact the effectiveness of ICP. Parallelization overhead falls in the first category, while CDR and microarchitectural characteristics (IPC, L2MI, etc.) fall in the second category.

We use hardware performance counters to measure all microarchitectural features. For PO and CDR, we use simple profiling. PO computes the number of barrier or lock executions per instruction while executing with one thread. A high PO implies the loop is unlikely to scale well. We also assign a large PO value to the loops that have limited software parallelism and loops where the computation increases linearly with the number of threads. These special loops also scale poorly with additional threads.

CDR is the percent of data that the main thread chunk shares with the helper thread chunk, when the application runs with one main thread. Write sharing is more expensive than read sharing (Section 4.3) because of the expensive cache to cache transfers. Thus, we give it more weight and calculate data reuse as: $read\_sharing + k \times write\_sharing$. The value of $k$ depends on how expensive coherence misses are. In our setup, any value of 2 or above works well. CDR varies from 0 to 50% for our loops.

We evaluate our linear classifier in two steps – first we try to find out the features that have strong correlation, and then we measure the effectiveness of the best linear classifier.

**Correlation of loop characteristics with the classification**

We analyze the weight vector of the linear classifier to understand the correlation of the loop characteristics. The magnitude of the weight defines how strong the correlation is. On the other hand, the sign indicates whether the correlation is positive or negative. In our case, positive correlation indicates a bias toward applying par-ICP.

To analyze the correlation, we construct the linear classifier using all 108 loops and 12 characteristics and compute the weight vector. We do 10-fold cross validation to avoid any bias and take the average weight over all 10 classifiers. For 10-fold cross validation, the data is partitioned into 10 subsets. Each classifier uses a different subset for testing while the remaining 9 subsets are used for training. The cross validation accuracy is the percent of data that are correctly classified. The cross validation accuracy in this case is 79%. There are few parameters (e.g., cost for incorrect classification) which we can tune during the training phase of the classifier. For all of our experiments, we tune these parameters to get the best accuracy.

Figure 5.3 shows the weight of the characteristics in the classifier. The strong positively correlated characteristics are PO, TLBL2IN, and L2MI, while CDR, DTLBIN, BRANCH exhibit negative correlation. Thus, a loop that has large PO, L2MI, TLBL2IN but low CDR, DTLBIN is most likely to gain from par-ICP. This figure demonstrates several interesting things. First, IPC and different types of stalls (fetch, dispatch) have negligible correlation with the decision process. These are complex metrics and depend on several things like branch misprediction, cache misses, data dependency, resource conflict, etc., but ultimately they do not provide concrete information useful for the classification.

BRANCH has a negative correlation because a high rate of branches indicates irregular code where it is difficult to prefetch the correct data – e.g., they are more likely to contain control-dependent loads or load addresses. L2MI and

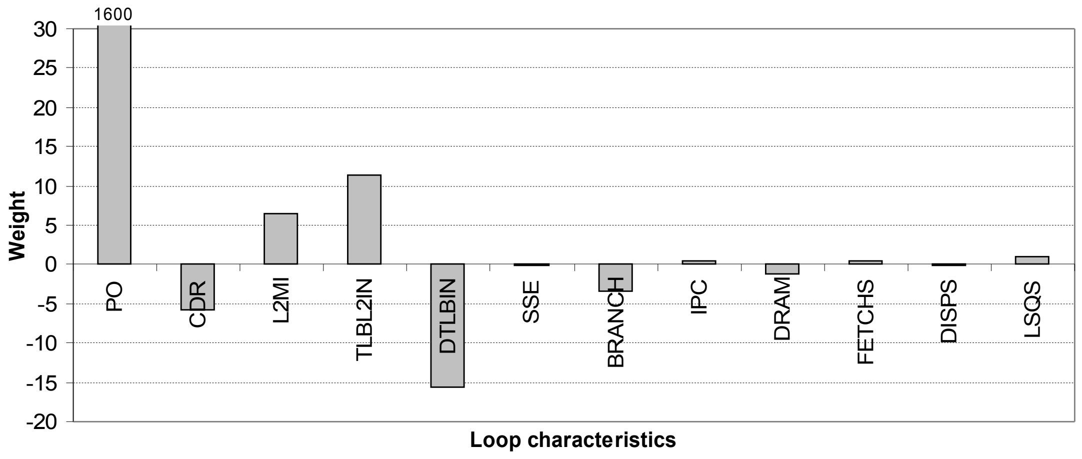


Figure 5.3: Weight vector generated by the linear classifier that represents correlation.

TLBL2IN have strong positive correlation since ICP primarily converts L2 misses into L2 hits. DRAM and DTLBIN have negative correlation. This is a little surprising since we expect all types of misses to have positive correlation. This is impacted by some cases where the loops are very memory intensive and the data sets are very large, requiring multiple helper threads for ICP to be effective – threads which are better used for traditional parallelism in those cases.

As expected, PO limits the scalability of traditional parallelization, and has strong positive correlation. Similarly, CDR reduces the effectiveness of ICP and has negative correlation.

Since several of the parameters have little impact on the outcome of the classifier, we can simplify it (and the required profiling) by removing some of the metrics. We remove the five characteristics (SSE, IPC, FETCH, DISP, and LSQS) that have the smallest correlation and construct the linear classifier with the remaining seven characteristics. The 10-fold cross validation accuracy, in this case, improves to 84%; thus, ignoring the characteristics that are not heavily correlated removes the noise and builds a better linear classifier. On the other hand, skipping some of the important characteristics and picking others does decrease the accuracy, e.g., using PO, CDR, L2MI, SSE and BRANCH decreases the accuracy to 69%. Our best linear classifier uses 7 characteristics – PO, CDR, L2MI, TLBL2IN,

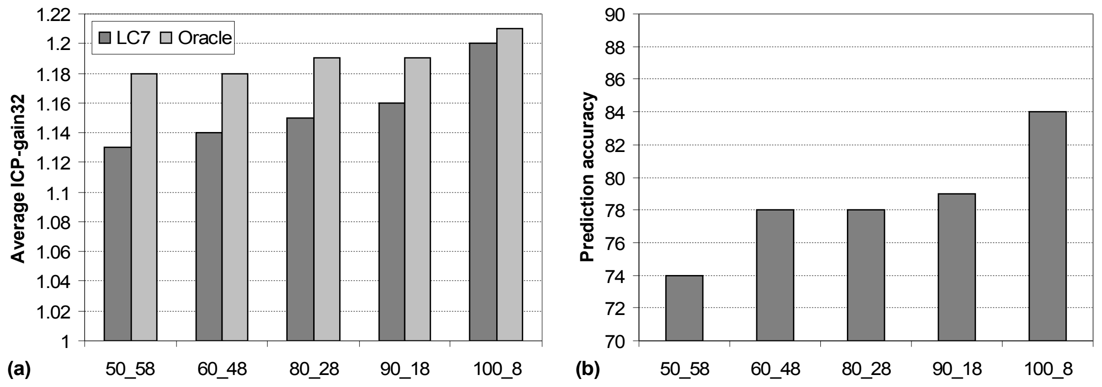


Figure 5.4: Average $ICP\text{-}gain_{32}$ (a) and prediction accuracy (b) for LC7 using different combinations of training and test set sizes (shown on the x-axis). The first number represents the training set size.

DTLBIN, DRAM, and BRANCH. We refer to this classifier as LC7.

## Linear classifier performance

This section measures the effectiveness of our best linear classifier, LC7, using standard evaluation techniques for machine learning. We randomly pick $k$ loops to train the linear classifier and use the rest of (108-k) loops to test it. While training the classifier, we assume nothing about the test loops and use cross validation for the train loops to avoid bias towards the training set. To cancel out the effect of randomization, for a particular training size of $k$, we repeat the procedure 20 times to pick different sets of training and test loops, and then take the average of the outcomes.

There are two metrics that we measure for the test set – average $ICP\text{-}gain_{32}$ and the percent of loops correctly classified in the test set, i.e. prediction accuracy. We compare our results with the oracle classifier which has prediction accuracy of 100%. Figure 5.4 shows the average $ICP\text{-}gain_{32}$ and prediction accuracy for different values of $k$. Here, the first value indicates the size of the training set while the second one indicates the size of the test set.

We use different training sizes from 50 to 100. The prediction accuracy improves with the train size, because the classifier can capture more information.

However, even if we use 50 loops for training and the other 58 loops for testing, we get very good accuracy of 74%. Considering the average *ICP-gain*$_{32}$, LC7 loses 5% performance compared to the oracle classifier. For a training set size of 100 loops, the accuracy improves to 84% and LC7 performs just 1.2% less well than the oracle.

We also do another test, where we use all 108 loops to construct the linear classifier and use that to predict the same 108 loops. In this case, the accuracy is 87% and LC7 provides around 1.4% less gain than the oracle classifier. This demonstrates two things – the loops are not linearly classifiable, so we can expect some error, and yet LC7 performs close to the oracle classifier.

For further information, we analyze the loops that LC7 correctly classifies. LC7 can more accurately classify the loops which are highly sensitive to the correct decision – the performance variation between applying par-ICP and traditional is large. If we only consider the loops where par-ICP either wins or loses by at least 5%, LC7 classifies 91% of loops correctly. The accuracy improves to 98% when we consider at least 25% performance variation. This is significant, because the linear classifier does not use any information regarding the possible performance gain or loss. This also explains why LC7 closely follows the oracle in terms of the average *ICP-gain*$_{32}$ despite selecting the less effective technique 13% of the time.

**Other Decision Heuristics** In an attempt to simplify the decision process for the compiler, we alternatively take the correlation data learned from the linear classifier, and use that to construct a simple decision matrix. Specifically, we are able to take just four of our highly correlated metrics (PO, CDR, L2MI, and DTLBIN), apply an experimentally derived threshold for each, and classify each loop according to whether it is high or low with respect to these four metrics. By doing so, we are able to replicate the accuracy of the linear classifier. The advantage of this construction is that it uses only four profile measurements and simple table lookup. However, since it does not provide gains above the linear classifier, we do not show the specific results here.

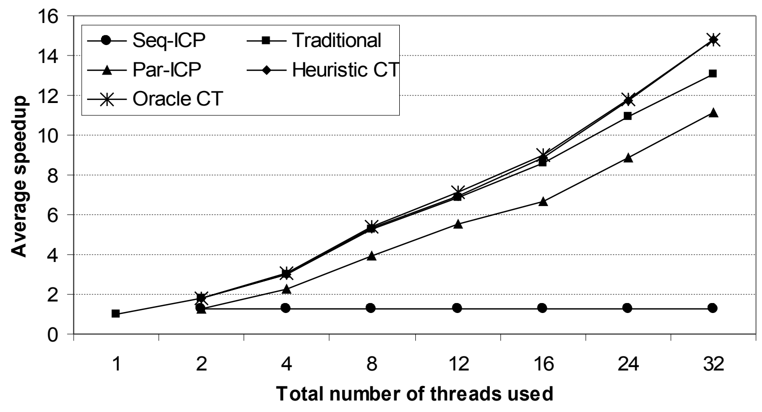


Figure 5.5: The scalability of the speedup averaged over all benchmarks using different techniques.

### 5.4.2 Application Level Impact

This section analyzes the impact of applying coalition threading (CT) to the applications. There can be several dominant loops in an application, so we have four different schemes to consider – apply seq-ICP to all loops, apply traditional parallelism to all loops, apply par-ICP to all loops and apply the combination of traditional parallelism and par-ICP as suggested by our heuristic. We call the last scheme *heuristic CT* which applies par-ICP to the set of loops chosen by the heuristic and applies traditional threading to the rest. For heuristic CT, our coalition threading framework uses the LC7 heuristic described earlier. We also implement *oracle CT* that applies coalition threading using the oracle heuristic to evaluate the performance of the LC7 heuristic at the application level.

Figure 5.5 shows the effect of applying all the schemes on our benchmarks in the Opteron system. The y-axis here denotes the average speedup over all benchmarks for a particular number of threads. The seq-ICP scheme cannot compete with other schemes, because most of them scale with traditional threading, whereas seq-ICP scales only marginally beyond 2 cores. Par-ICP scales better than traditional threading and gradually catches up – 28% performance loss for 2 threads vs. 15% loss for 32 threads. The impact of par-ICP is more prominent in case of heuristic CT. For all thread counts, heuristic CT outperforms traditional

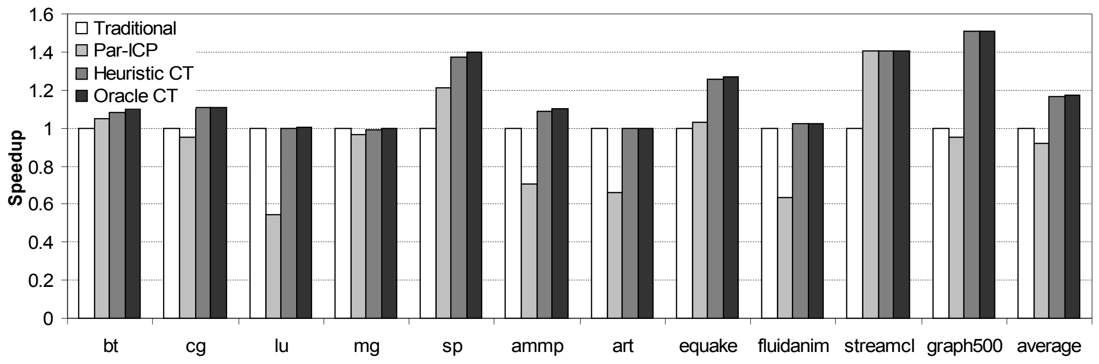


Figure 5.6: The graph compares speedup (normalized to best traditional performance) of oracle CT and heuristic CT with other techniques in Opteron. The improvement is as high as 51%.

threading. The improvement is 1% for 4 threads, but reaches 13% for 32 threads. As expected, the utility of coalition threading increases with core count. There is no reason to expect this trend does not continue. Figure 5.5 also shows the strength of the LC7 heuristic. It performs almost the same as the oracle heuristic.

Next, in Figure 5.6, we compare the performance of heuristic CT with par-ICP, traditional threading, and oracle CT across all benchmarks. The values on the y-axis are normalized to the best possible speedup by traditional threading using no more than 32 cores.

In the Opteron machine, we see gain as high as 21% for *SP* and 40% for *streamcluster* while using the par-ICP scheme. Oracle CT improves performance further from 21 to 40% for *SP* by filtering the loops that do not get any benefit from par-ICP. However, for *streamcluster*, both the key loops get benefit from par-ICP and oracle CT gives the same speedup as the par-ICP scheme. *Graph500* shows the importance of using a good heuristic. For this application, heuristic CT provides 51% speedup vs. the 5% loss by par-ICP. Overall, the oracle CT gives an average of 17.4% gain across all benchmarks, and an average gain of 21% considering only the nine benchmarks where we apply par-ICP to at least one loop. Compared to that, heuristic CT gives an average of 16.7% gain across all benchmarks and performs very close to the oracle CT. The small number of loops where our heuristic does not make the correct decision have negligible impact on the overall application run-time. So our developed heuristic is nearly as effective as the oracle heuristic.

## 5.5 Conclusion

This chapter describes coalition threading, a hybrid threading technique that combines traditional parallelism with non-traditional parallelism, and demonstrates heuristics to find the best combination, which can be used to direct a parallel compiler. We analyze the effectiveness of coalition threading using inter-core prefetching as the non-traditional component and observe that for a number of parallel loops, adding ICP on top of traditional threading outperforms traditional threading alone. Our results show that in a 32-core Opteron system, there are 20 loops out of 108 where coalition threading provides more than 30% improvements on top of the best possible speedup by traditional threading. Coalition threading has a strong impact on application level scalability (up to 51% performance gain in some cases) when we apply the right technique to the right set of loops. Our best heuristic makes the correct decision 87% of the time and can accurately identify 98% of the loops where the correct decision is critical. Heuristic-based coalition threading gives an average of 17% improvements across our benchmarks and is only 0.7% less than what an oracle provides.

## Acknowledgments

Portions of this research were funded by the US National Science Foundation grants CCF-1018356 and NSF-0643880 and by SRC Grant 2086.001.

This chapter contains material from “Coalition Threading: Combining Traditional and Non-Traditional Parallelism to Maximize Scalability”, by Md Kamruzzaman, Steven Swanson and Dean M. Tullsen, which appears in PACT’12: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, September 2012. The dissertation author was the primary

investigator and author of this paper.

# Chapter 6

# Underclocked Software Prefetching

In last few chapters, we show the techniques that improve application performance by accelerating each single thread using multiple cores. However, for today's computing platforms, in many scenarios, it is of equal or greater importance that we should be able to exploit multiple cores to reduce energy or improve energy efficiency (e.g., energy-delay product).

This chapter shows the potential of inter-core prefetching (Chapter 4) to provide such opportunity. Using helper thread techniques on a multicore, not only can we decrease runtime for serial code, but we can also decrease consumed energy. Here, we demonstrate a helper-thread runtime system, which is uniquely constructed to exploit higher core counts for decreased energy and to manage the complex energy/performance/core count tradeoffs. We call this new technique underclocked software prefetching.

In fact, non-traditional parallelism (e.g., helper thread based parallelism, where parallel speedup is achieved without dividing up the original computation) shows new directions for energy optimization. Most parallel code creates homogeneous threads, leaving little opportunity for per-thread energy optimization. Helper thread parallelism is inherently heterogeneous, meaning each thread can be optimized separately (e.g., running the main thread at a high frequency and helper threads at a low frequency). That heterogeneity exists across two dimensions. First, the main thread is compute intensive while the helper thread is memory-intensive, meaning their tolerance to frequency scaling is likely to differ. Second, they are typically imbalanced, allowing us to exploit slack by scaling the frequency of a fast helper thread.

Inter-core prefetching (ICP) enables the helper threads to execute on distinct cores, yet still target the L1 cache, on existing architectures. It does so by migrating the main thread to a core for which the data has already been prefetched by the helper, simultaneously migrating the helper thread to another core to preload new data into that cache. In this way, the main thread is constantly executing in a core for which the current data working set is already present in the data cache, regardless of the size of the working set. Because helper threads execute in different cores, their frequency and, in the future, voltage can be decoupled from that of the main thread.

The combination of inter-core prefetching and per-core frequency control creates several unique opportunities. First, with the helper thread in a separate core, we can manage the power of the main thread and the helper threads separately, exploiting the natural heterogeneity. Second, this allows us to minimize the incremental power consumed by the helper threads, while preserving the full performance gain of ICP, resulting in dramatic reductions in energy. Third, it has been shown that ICP easily handles slow helper threads by simply using more cores – thus, the complete spectrum of core count vs. per-core power can be exploited to find the optimal operating point.

This particular work shows that ICP itself can decrease energy over traditional execution, by a factor of two. Additionally, by controlling the frequency of helper threads, we have the potential to match the performance of the best ICP result, while further cutting the energy by as much as 47%. However, we also show that this potential is not realized on current systems, as ICP is very sensitive to the (currently very high) latency of changing frequency. We show that future systems with lower latency, however, will be able to exploit this technology.

The rest of this chapter is organized as follows. Section 6.1 discusses processor power management, and the details of the technique. Section 6.2 describes our methodology. Section 6.3 evaluates the technique, and Section 6.4 concludes.

## 6.1 ICP with Frequency Scaling

This section describes how we apply frequency scaling to maximize the energy-efficiency of ICP. We refer to this approach as *ICP-dynamic*, where we apply per-core dynamic frequency scaling, potentially allowing different frequencies for the main thread and the helper threads. The optimal use of dynamic frequency control depends heavily on the underlying capabilities and properties of the system, so we first give a brief overview of processor power and frequency management in a state of the art system, and then analyze the impact of frequency scaling on applications. Following that, we describe how to apply ICP-dynamic to single thread execution.

### 6.1.1 Processor power management

Most modern processors support multiple performance states (P-states) to manage power consumption. Each P-state is characterized by an operating voltage and frequency. A lower P-state means the processor operates at a lower voltage and frequency. Power consumption follows the equation $P \propto V^2 f$ where $P$, $V$, $f$ stand for power, voltage, and frequency, respectively. So, there is a considerable increase in power consumption when the processor moves into a higher P-state.

In the case of multicore processors, different systems allow different levels of heterogeneity – same voltage and frequency on each core, same voltage but different frequency, or different voltage and frequency settings per core. Systems like Intel Nehalem require the same voltage and frequency for all active cores, whereas AMD systems (starting from generation 0x10) allow cores to run in different frequency but with the same voltage. This is known as “Independent Dynamic Core” technology in AMD terminology. There are no general-purpose processors yet that allow different voltage and frequency settings per core. Therefore, we use per-core dynamic frequency scaling in this study – future systems that allow per-core DVFS will see greater improvements in energy efficiency than what we measure

Table 6.1: Operating voltage and frequency for different P-states in the AMD Phenom used in our experiments.

| **P-state** | **Voltage (V)** | **Frequency (MHz)** |
|---|---|---|
| P0 | 1.25 | 3,100 |
| P1 | 0.90 | 2,600 |
| P2 | 0.85 | 2,000 |
| P3 | 0.76 | 1,400 |
| P4 | 0.76 | 800 |

and project in these results.

The AMD Phenom processor supports five P-states including one turbo state. Table 6.1 shows the different P-states for the AMD Phenom system that we use. Here P0 is the turbo state and it is only available when at least half of the total cores are inactive. AMD Phenom allows different P-states for different cores. In this case, all cores will operate at the voltage of the highest P-state, but will run at different frequencies. So, if core-0 is in P2, and core-1 is in P4, both cores will operate at 0.85V, but at 2000 MHz and 800 MHz frequency, respectively. This consumes less power than running both cores in P2, but uses more power than using different voltages.

### 6.1.2 Impact of frequency scaling

Applications show different sensitivity to changing P-states (i.e. frequencies). Figure 6.1 shows the impact on performance using different P-states for two different kernels – one cpu-intensive, and one memory-intensive. As expected, the memory-intensive kernel is less sensitive to the change of frequency. If we change the frequency from 800MHz to 3100MHz (3.9$\times$ increment), performance only improves by 2.1$\times$. This happens for two reasons. First, the memory subsystem runs in a different (and slower) frequency domain than the cores, so changing core frequency does not impact the performance of the memory. Second, the cpu stalls frequently while executing memory-intensive kernels, and reducing the frequency decreases the latency of cpu stalls (measured in processor cycles). For cpu-intensive kernels, performance is highly proportional to the frequency.

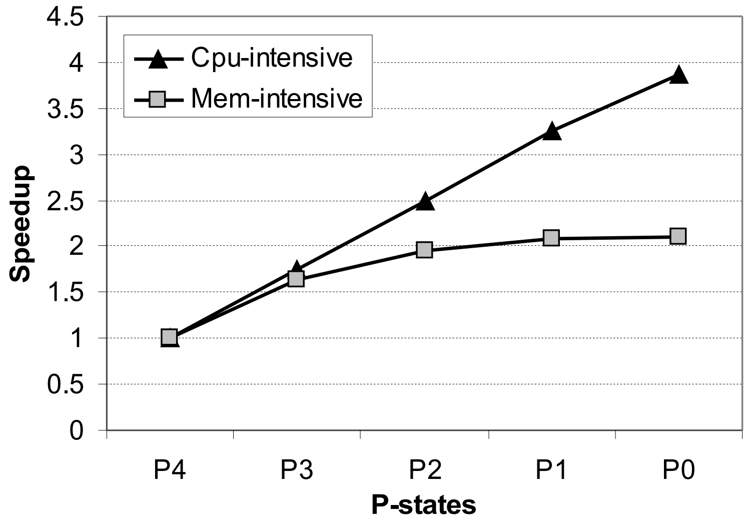


Figure 6.1: Speedup for cpu-intensive and memory-intensive kernels in different P-states.

Decreasing frequency makes an application run longer, but at lower power, so the impact on energy is unclear. Additionally, two competing factors make the problem even more complex. First, higher frequency requires higher voltage so there can be a cubic increase in the power consumption. This favors lower frequency. On the other hand, other system components, including the memory hierarchy, consume a large amount of fixed power irrespective of whether the processor is executing or not, and this favors executing as fast as possible. Current systems are not energy proportional and the latter effect tends to dominate. Therefore, for cpu-intensive applications, the most performance efficient execution is usually also the most energy efficient. However, the scenario changes for the memory intensive applications. Prior works address these issues [HK03, KGyWB08, CSP04] and show the effectiveness of DVFS for energy efficiency.

These insights impact our results in several ways. First, ICP transforms the main thread from memory-intensive to cpu-intensive. Second, the helper threads themselves are memory intensive; however, because our helper threads exhibit much higher memory-level parallelism than typical code, their forward progress scales with frequency somewhat more than typical memory-intensive code.

### 6.1.3 ICP and frequency scaling

ICP provides an opportunity to apply per-core dynamic frequency scaling to make single thread execution more power and energy efficient.

ICP decouples the memory intensive prefetching stream from the execution stream. The prefetching stream is less sensitive to core frequency, and can execute in parallel. Thus, we can use multiple prefetch threads to decrease the average prefetching time. Conversely, the execution stream is more sensitive to core frequency, and is strictly sequential. Thus, the only way to accelerate the main thread (after ICP is already applied) is to use higher frequency.

ICP-dynamic has the ability to use different frequencies for the prefetching and execution streams. However, since the main thread and helper threads swap cores at frequent intervals, we cannot use a fixed frequency for each core. Instead, we need to change frequency dynamically when a core switches from prefetching to execution or vice versa. This makes ICP-dynamic highly sensitive to the latency of changing frequency, which on current systems is very high.

We can explain the impact of ICP-dynamic (ignoring for now the latency of frequency changes) using the following equation:

$$T = max(e, (e + p)/n) \tag{6.1}$$

Here $T$ is the average time to process a chunk using ICP, $e$ is the time to execute (after data is already prefetched) a chunk by the main thread at some frequency, $p$ is the time required to prefetch data for a chunk by the helper thread at some frequency, and $n$ is the total number of cores used – e.g., in the case of one helper thread, $n$ is 2. This equation holds whether the main thread and helper threads use the same or different frequency.

Equation (6.1) demonstrates several things. The theoretical maximum speedup of ICP is $b/e$ where $b$ is the time to execute a chunk without ICP. When $e$ is larger than $p$, we only need one helper thread to get the best speedup. In this case, the helper thread waits $e - p$ to synchronize with the main thread. We can lower the operating frequency for the helper thread while $e \geq p$. This will reduce power consumption without sacrificing any performance benefit. When $p$

Table 6.2: Memory hierarchy information, migration cost, and the latency to do frequency/voltage scaling for AMD Phenom.

| **CPU components** | **Parameters** |
|---|---|
| CPU model | AMD Phenom(tm) II X6 1035T |
| Number of cores | Single-socket 6 cores |
| Level-1 (L1) cache | 64-Kbyte private cache, three cycles |
| Level-2 (L2) cache | 512-Kbyte private cache, 15 cycles |
| Level-3 (L3) cache | 6-Mbyte shared cache, 40 cycles |
| Memory | 8 Gbytes, 120 cycles |
| Latency to frequency change | $4 - 9\mu s$ |
| Latency to voltage and frequency change | $14 - 70\mu s$ |
| Idle state power consumption | 94.5W |

is larger than $e$, a more complicated case arises. Lowering the frequency of either stream will hurt performance in this case. However, we can reduce the frequency of the prefetching stream if we increase the number of prefetch threads. A lower frequency increases $p$, but increasing $n$ easily offsets that since prefetching is less sensitive to frequency change. Meanwhile, using a lower frequency reduces per core power consumption.

*ICP-static* is an alternative option. In this case, core frequencies remain fixed over time but different cores may operate at different frequencies. Threads in this case alternate in different frequencies. ICP-static is not typically a great option compared to ICP-dynamic or *ICP-same* (all cores at same frequency), but could be used to keep the power consumption under some budget.

## 6.2 Methodology

This section describes our experimental methodology – the application kernels that we target, and the systems that we use to evaluate ICP-dynamic and ICP-static.

### 6.2.1 Evaluation Systems

We use a 6-core AMD Phenom desktop system running Ubuntu Linux 2.6. Table 6.2 gives more information about the system. From Table 6.2, it takes more time to change voltage than just frequency. The time also scales with the difference between the P-states. For example, when a core switches from P0 (1.25V, 3100MHz) to P4 (0.76V, 800MHz), it takes around $70\mu s$. However, switching to P2 (0.85V, 2000MHz) from P0, it takes around $14\mu s$. The core is unavailable during the P-state switching time.

We compile all codes using gcc version 4.5.2 with -O3 optimization level and keep hardware prefetching enabled for all experiments. We do all power measurements using a "Watts up? .Net" power meter [Dev]. It measures the power for the system at the wall and reports power consumption in 1-second intervals with $\pm 1.5\%$ accuracy.

We read and write the model-specific registers using privileged instructions *rdmsr* and *wrmsr* to change the P-state of a core.

### 6.2.2 Application kernels

We extract kernels from 11 applications from several different benchmark suites – *bt, cg, mg, lu, sp* from NAS, *equake* from Spec2000, *lbm, libquantum, mcf, milc* of Spec2006, and *svm-rfe* of Minebench. All of these kernels are key processing loops of the applications and dominate their performance. We exclude benchmarks from these suites that are not written in C/C++ or are not memory-intensive. Using kernels instead of the full application enables us to keep the power consumption steady for more accurate measurement.

## 6.3 Results

We evaluate frequency scaling for ICP in several steps. First, we baseline the power consumption behavior of the system. Then we measure the performance improvements and power consumption of our kernels using ICP-same. Third, we

Table 6.3: Power consumption for different P-state combinations. Using one core in P1 and two cores in P3 will consume $106.8 + 2 * 4.7 = 116.2W$.

| **Core-0 P-state** | **Power consumption (W)** | **Additional core power (W)** | | | | |
|---|---|---|---|---|---|---|
| | | **P0** | **P1** | **P2** | **P3** | **P4** |
| P0 | 125.6 | 13.8 | | 7.5 | 6.1 | 3.3 |
| P1 | 106.8 | | 8.4 | 6.6 | 4.7 | 2.9 |
| P2 | 102.8 | | | 6.0 | 4.4 | 2.6 |
| P3 | 98.1 | | | | 3.9 | 2.3 |
| P4 | 96.6 | | | | | 2.2 |

apply ICP-dynamic and ICP-static and measure the performance, power, and energy. Finally, we simulate the effect of lowering the latency of frequency or voltage scaling.

In this section, we will use the term *active power* to represent the power that we get by subtracting idle power from the power measured at the wall. So, if the system uses 120W power, the active power is $120 - 94.5 = 25.5$W. Similarly, we can compute *active energy*.

### 6.3.1 Power consumption for different P-states

We measure the power consumption for different P-state combinations to understand how it varies when P-state changes. For this experiment, we use a simple loop doing arithmetic operations and no memory accesses. Table 6.3 shows the power consumption for different combinations. The 2nd column represents the power when one core is active, and columns 3–6 represent the additional power required when we activate one more core at a particular P-state. So when two cores run at P0, the power consumption is $125.6 + 13.8 = 139.4$W.

Table 6.3 shows three things. There is a large jump in active power consumption when we increase the voltage (active power is 2.1W at P4, 31.1W at P0). Second, as expected the incremental power of adding a core depends on the P-state of the first core, since the first core (the one at higher P-state in this table) determines the voltage. Third, it takes more power to use a high P-state core rather than multiple low P-state cores. For example, 2 cores at P0 will consume

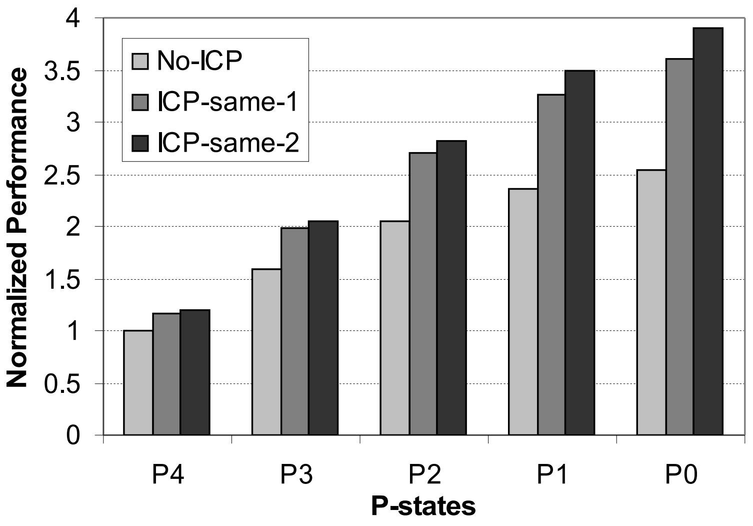


Figure 6.2: ICP-same speedup in different P-states. Here 1, 2 represent the number of helper threads.

139.4W, but one in P0 and 3 cores in P4 will consume $125.6 + 3.3 \times 3 = 135.5$W. This last characteristic is an opportunity for ICP since we can often replace one fast prefetch thread with multiple slower ones with no loss in performance.

### 6.3.2 ICP without frequency heterogeneity

ICP provides significant performance improvements for memory intensive kernels. Figure 6.2 shows the speedup averaged over all kernels using traditional ICP (ICP-same) and without using ICP (No-ICP) at different P-states. We use a chunk size of 128K and apply ICP with both 1 and 2 helper threads.

From Figure 6.2, the effectiveness of ICP increases with higher P-states (higher frequency). The processor-memory gap accounts for this. Cache misses are more expensive in P0 than in P4, so eliminating those profits more. We also see that ICP continues to improve performance linearly with frequency whereas without ICP, performance tapers off at higher frequency. This is expected – without ICP, most of these applications are memory bound; with ICP, they become compute bound.

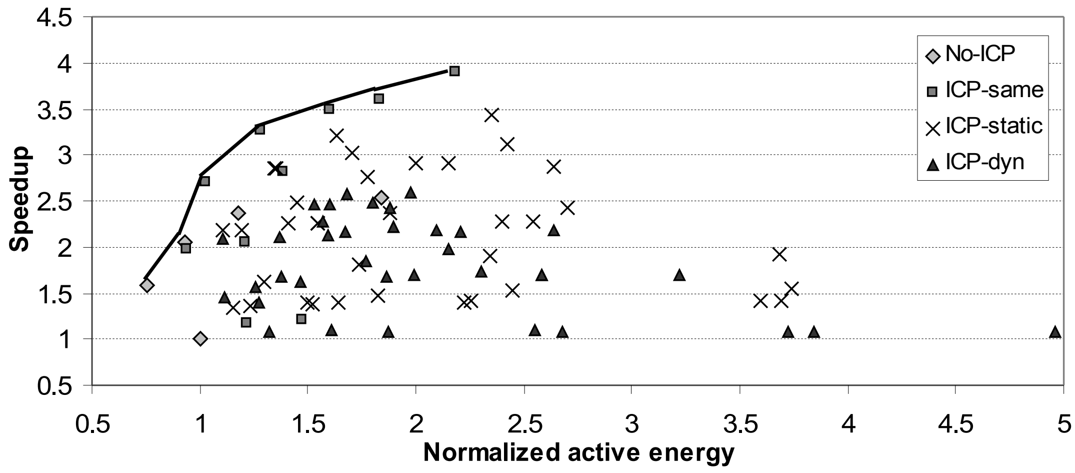


Figure 6.3: Scatter plot of normalized speedup and active energy consumption averaged over all kernels for different P-state combinations. The points on the line are pareto-optimal.

### 6.3.3 Performance of ICP-dynamic and ICP-static

We compare ICP-same (all cores at same frequency), ICP-static (cores different in frequencies, but do not change, whether running main thread or helper thread), and ICP-dynamic (cores change frequencies when transitioning from main thread to helper thread and back) on our AMD system. We compute the power measurements and performance for all possible P-state combinations for all of our kernels. We use 128K chunks and vary the number of helper threads between 1 and 2. In each case, we measure the power consumption, active power, energy, and active energy.

Because there are so many possible P-state combinations, we show a scatter plot of the possible combinations in Figure 6.3, plotting active energy and speedup. The pareto-optimal points (connected by the line) are the interesting ones – those where no other single data point is both above and to the left. From this figure, we see that none of ICP-static or ICP-dynamic points are pareto-optimal and ICP-same works best – six out of seven pareto-optimal points are ICP-same. Looking more closely, some of the ICP-dynamic points are pareto-optimal for individual kernels, but not in general across all kernels. This is a direct result of the high latency to switch frequency and/or voltage. Recall from Section 6.1.3, ICP-dynamic performs two P-state changes per chunk and in some cases, we see a nasty side effect of the chip P-state policy – consider the case where the main thread is at P1 and the prefetch threads are at P3. In that case, the main thread determines the chip voltage. But, if the core running the main thread finishes early, it starts a prefetch thread and the processor will seek to change the voltage because all threads are temporarily at P3, resulting in an expensive transition.

Increasing chunk size reduces the frequency of P-state changes. However, using chunk size larger than 512K (the size of the L2 cache) will drastically impede ICP performance. We run the above experiment with 512K chunk size and find that still none of the ICP-dynamic points are pareto-optimal. The average switching interval in P0 becomes $100\mu s$ compared to $24\mu s$ with 128K chunk and the performance drop is around 15%. On the other hand, the P-state change latency varies from $4 - 70\mu s$, the same order of magnitude as the chunk execution time in some cases. We expect a drop in the future voltage and frequency change latency, which will change these results significantly.

In the next section, we further evaluate the scenario when the latency to change P-state is reduced.

### 6.3.4 Sensitivity to P-state change latency

ICP-dynamic does not perform to its potential because of the high P-state change latency in our experimental system. Recent research [KBW12] shows the potential for nanosecond level voltage/frequency switching. The authors report a voltage transition between 4 and 1.4V within 20ns. Also, systems like IBM Power7 [FAWR$^+$11] can gradually change core clock frequency while the core is fully operational – it has essentially zero P-state change latency.

We build a simulation model that analyzes the impact of varying P-state change latency. We develop the model using the timing and power consumption data collected from our AMD Phenom system. The model requires six values for each P-state – time required to execute a chunk without ICP, time to prefetch data for a chunk, time to execute a chunk when the data is already in the cache, and

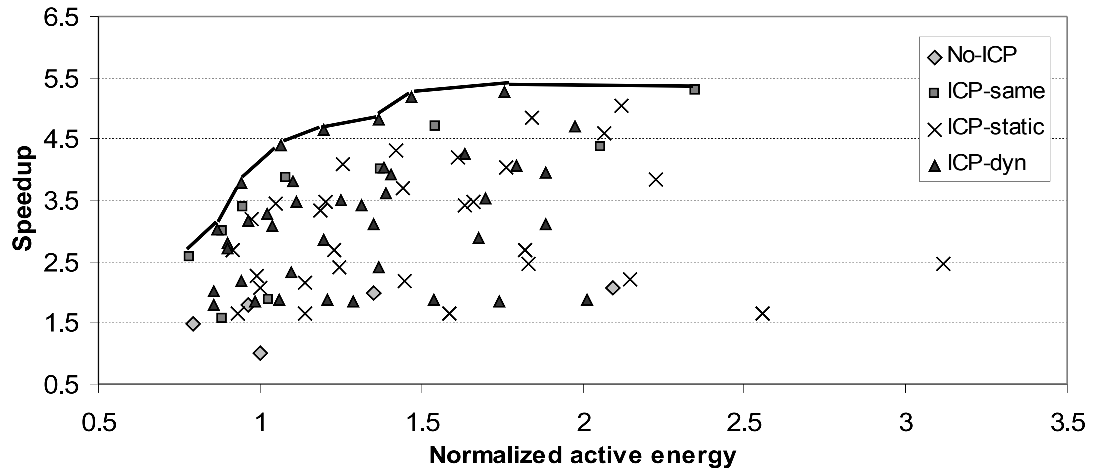


Figure 6.4: Scatter plot of normalized speedup and active energy consumption assuming zero latency to change P-states. Pareto-optimal points are shown on the line.

power requirements in the previous three cases. We feed this matrix to our model, and it computes for each core the amount of time the core prefetches, executes, and spins for each P-state combination. For ICP-dynamic, the model also considers the latency to change P-states. The model needs two additional power consumption values – when the core is spinning, and when it is switching to a new P-state. The first is easy to measure. For the second, we take the average power consumption of the participating P-states. The model works for both ICP-dynamic and ICP-static, and handles any number of helper threads.

For a particular P-state combination, our model computes the power consumption of each participating core, and the average processing time of a chunk ($T$ in Equation (6.1)), which we again use to compute the performance gain, energy, or energy-delay product (EDP). We denote a P-state combination by $Pxy$ meaning that the main thread runs at Px, while the helper thread runs at Py. So, P00 refers to ICP-same at P0 while P03 indicates ICP-dynamic with main thread at P0 and helper threads at P3.

Figure 6.4 shows the scatter plot similar to Figure 6.3 (128K chunk, 1–2 helper threads) assuming no latency to change P-states. ICP-dynamic is pareto-

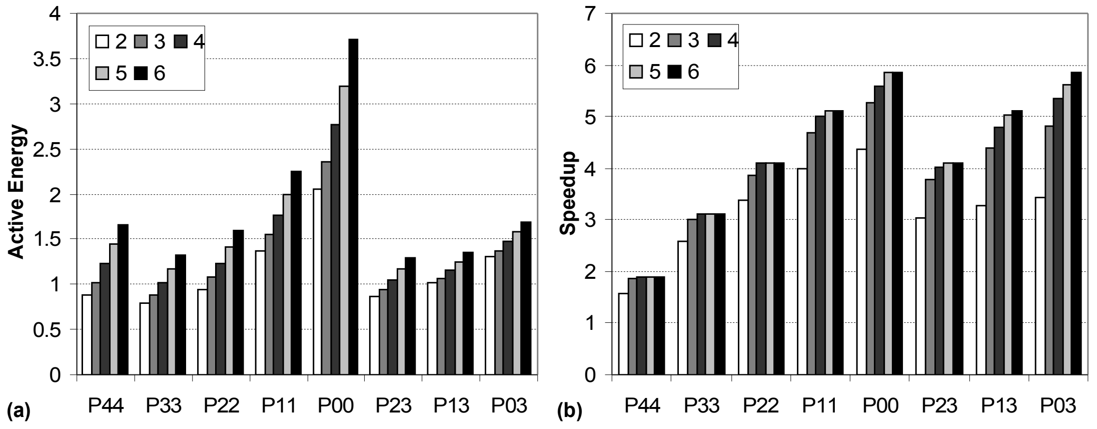


Figure 6.5: The impact of adding more cores for different P-state combinations – (a) Active Energy, (b) Speedup. Here Pxy means the main thread is in state Px and the helper threads all operate in state Py.

optimal with respect to all other techniques except P33 and P00. P33 takes the least energy of all ICP combinations, but the performance of that point is quite low. P00 provides a little more speedup at the expense of much larger energy. If we consider other metrics, such as EDP or active EDP, ICP-dynamic dominates those two points as well.

At the top of Figure 6.4, we see that ICP-dynamic comes within 1% of the maximum (P00) speedup (5.28×), using the P01 combination which consumes 26% less active energy. Alternatively, we can get within 2% with P02 while consuming 38% less active energy.

We can underclock the helper threads further while use more cores to attain the same speedup as ICP-same. In our experiments, P3 seems to be the best choice for helper threads. Figure 6.5 shows the impact of varying the number of helper threads for the five ICP-same and three ICP-dynamic choices – P03, P13, and P23. We use up to 5 helper threads (6 cores). The graph shows the advantage of using underclocked prefetchers. P00 reaches the theoretical maximum speedup by ICP using 5 cores. P03 eventually catches up to the speedup provided by P00 using 6 cores, but consumes 47% less active energy. Similarly, P13 consumes 32% less active energy and still provides the same speedup as P11. ICP-dynamic is effective

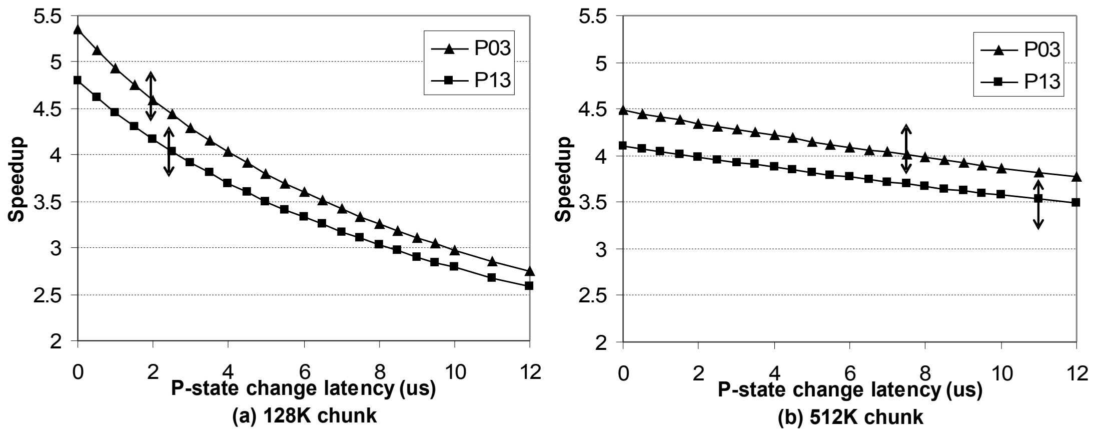


Figure 6.6: The Impact of modifying the latency of P-state change on P03 and P13.

in saving energy with no sacrifice in performance even over the most aggressive ICP scheme.

We have shown results to this point with both the full frequency switch latency, and no latency. Figure 6.6 shows the change in speedup for P03 and P13 when the latency varies from 0 to $12\mu s$. We present the results for both 128K and 512K chunk size to better understand the sensitivity. Note that, given the granularity of this graph, the most accurate estimate of expected future switch times (in the few ns [KBW12]) is actually the 0 point. The vertical line shows the upper limit of the latency beyond which the ICP-dynamic point would no longer be pareto-optimal. For 128K chunks, this happens around $2\mu s$ of latency. Use of 512K chunk tolerates much higher latency. From Figure 6.6(b), P13 stays pareto-optimal until $11\mu s$. However, the performance drop is significant for 512K. If we consider both the 128K and 512K set of points in our scatter plot, none of the 512K points become pareto-optimal for any P-state change latency. So, the additional performance provided by the smaller chunks is critical.

Our model also allows us to predict the effects of per-core voltage scaling. With the same P-states, zero latency switching, and the ability to vary voltage across cores, P03 can achieve the same speedup as P00, consuming 54% less active energy (frequency scaling alone achieves 47%).

## 6.4 Conclusion

This chapter demonstrates techniques that enable the use of more processor cores to both improve performance and energy efficiency, even on completely serial code. It combines a multicore-based helper thread technique with explicit control of core frequency. The ability to manage the power consumption of the helper thread enables the system to minimize the incremental power of the helper thread, without compromising the resulting performance gains. This results in heavy gains in both energy and energy-delay product.

Furthermore, we showed that the long latency for frequency switching on current systems is a barrier to full realization of this technique, but future systems with faster switching will be able to benefit greatly. Compared to the state of the art helper thread prefetcher, we can match the same performance while consuming half the energy.

## Acknowledgments

Portions of this research were funded by the US National Science Foundation grants CCF-1018356 and NSF-0643880 and by Semiconductor Research Corporation Grant 2086.001.

This chapter contains material from “Underclocked Software Prefetching: More Cores, Less Energy”, by Md Kamruzzaman, Steven Swanson and Dean M. Tullsen, which appears in IEEE Micro, July-Aug. 2012, Volume 32, Issue: 4, Page(s): 32-41. The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2012 by the Institute of Electrical and Electronics Engineers (IEEE). Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

# Chapter 7

# Load-Balanced Pipeline Parallelism

The techniques discussed so far in this dissertation accelerate single thread execution without dividing up the computation. So, the potential gain is somewhat limited by the latency variations in different levels of memory hierarchy, or program internals like how memory intensive the application is, data access pattern, etc. In this chapter, we focus on extracting the parallel threads that will do computations, so that we can scale sequential performance as we add more cores. We look at the applications that are difficult to parallelize due to data dependences in the key loops, making the code highly sequential.

Prior work on *decoupled software pipelining* [RVVA04, ORSA05, ROR$^+$08] addresses the same problem and shows that fine-grained pipeline parallelism applied at the loop level can be very effective in speeding up some serial codes, including irregular codes like pointer chasing. In this case, the compiler automatically divides the loop into a set of pipeline stages (each stage can be sequential or parallel) and maps them to different cores to achieve parallel execution while still maintaining all the dependencies. However, several issues make the technique less attractive in practice. First, the cores often remain underutilized because of the imbalance in the pipeline, sacrificing performance and wasting energy. Second, the technique sacrifices the existing locality between stages, communicating data that was originally local across cores, again sacrificing both performance and power/energy. Finally, the technique typically works best with the number of threads at least equal to the number of stages, requiring the compiler to know a priori the number of cores available, and causing inefficient execution when the counts do not match.

This chapter describes *load balanced pipeline parallelism* (LBPP) which exploits the same pipeline parallelism as prior work, but assigns work to threads in a completely different manner, maintaining locality and naturally creating load balance. While prior pipeline parallelism approaches execute a different stage on each core, LBPP executes all the stages of a loop iteration on the same core, but achieves pipeline parallelism by distributing different iterations to the available cores and using token based synchronization to handle sequential stages. It groups together several iterations of a single stage, though, before it moves to the next stage.

LBPP is inherently load-balanced, because each thread does the same work (for different iterations). It maintains locality because same-iteration communication never crosses cores. The generated code is essentially the same no matter how many cores are targeted – thus the number of stages and the thread count are decoupled and the thread count can be determined at runtime and even change during runtime. Prior techniques must recompile to perform optimally with a different core count.

Like our other techniques, LBPP is a software only technique and runs on any cache coherent system. In this chapter, we demonstrate a compiler and runtime system that implements LBPP. We find that *chunking* is the key to making LBPP effective on the real machines. Chunking groups several iterations of a single stage together before the thread moves on to the next stage (executing the same chunk of iterations of that stage). Thus, chunking reduces the frequency of synchronization by clustering the loop iterations. Instead of synchronizing at each iteration, we only synchronize (and communicate) at chunk boundaries. Chunking appropriately allows us to maximize locality within a target cache.

In this chapter, we describe the implementation of LBPP under Linux and evaluate it on two multicore systems. We experiment with different microbenchmarks, focused on particular aspects of LBPP to characterize the technique better, and then apply it on a wide range of real applications (both regular and irregular). LBPP provides 1.7× to 3.2× speedup on AMD Phenom, and 1.6× to 2.3× on Intel Nehalem on average, depending on the core counts for individual loops. Considering application level speedup, some irregular applications see as large as 5.4× speedup over the single thread execution. We also compare LBPP with decoupled software pipelining. LBPP outperforms decoupled software pipelining by 60% to 88% on the loop level for a particular core count (three cores) because of better load balancing and locality. The energy savings can be more than 50% for some applications.

The remainder of this chapter is organized as follows: Section 7.1 describes the details of the technique. Section 7.2 shows the compiler implementation. Sections 7.3 and 7.4 demonstrate our methodology and results, and Section 7.5 concludes.

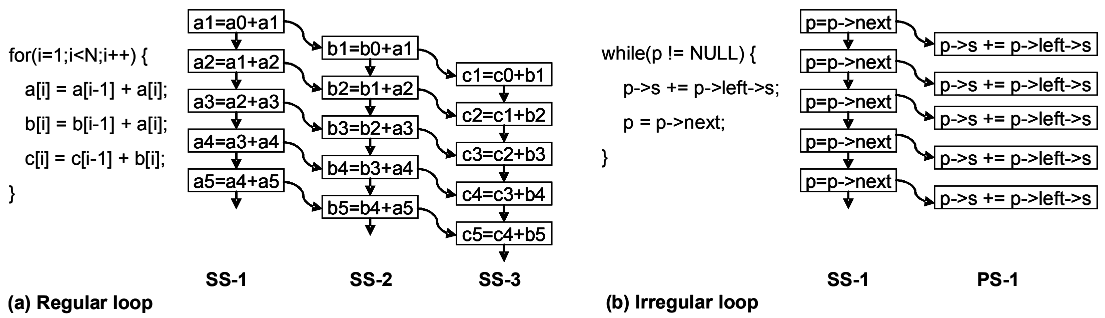
## 7.1 Load-balanced Pipeline Parallelism

Load-balanced pipeline parallelism exploits the pipeline parallelism available in the loops of applications and executes them in a data parallel fashion. It uses token-based synchronization to ensure correct execution of sequentially dependent code and handles synchronization overhead using chunking. The iteration space is divided into a set of chunks and participating threads execute them in round-robin order. The technique can handle both regular and irregular codes.

The following sections give an overview of pipeline parallelism and then describes LBPP.

### 7.1.1 Pipeline parallelism

Pipeline parallelism is one of the three types of parallelism that we see in applications (vs. data parallelism and task parallelism). Pipeline parallelism works like an assembly line and exploits the producer-consumer relationship. There are several pipeline stages and each stage consumes data produced by previous stages.



```
for(i=1;i<N;i++) {
   a[i] = a[i-1] + a[i];
   b[i] = b[i-1] + a[i];
   c[i] = c[i-1] + b[i];
}
```

```
while(p != NULL) {
   p->s += p->left->s;
   p = p->next;
}
```

Figure 7.1: Examples of loop level pipeline parallelism for regular and irregular loops. The arcs show the dependency relationships.

This maintains the dependency between different stages. The parallelism in this case is on the order of the number of stages. Pipeline parallelism can be coarse-grain or fine-grain.

In this work, we are interested in fine-grain pipeline parallelism extracted from loops in conventional code. Pipeline parallelism might exist in both regular and irregular loops. Figure 7.1 shows examples of pipeline parallelism in both types of loop. It also depicts the two different types of dependency relationships that exist between stages – *intra-iteration* dependency, and *cross-iteration* dependency. The former denotes the dependency between two stages in the same loop iteration whereas the latter represents a dependence to stages from previous iterations.

The regular loop in Figure 7.1(a) has three pipeline stages – SS1, SS2, and SS3. SS1 has no intra-iteration dependency (no incoming arcs from different stages), but has a cross-iteration dependency. SS2, and SS3 have both types of dependencies. All three stages in this case are *sequential* because of the cross-iteration dependency. This particular loop has a pipeline of three sequential stages. We can map each stage to a separate core and execute the stages in parallel while still maintaining all the dependencies. Prior works describe this technique as decoupled software pipelining [RVVA04]. The maximum possible parallelism for this loop is 3× assuming there are a sufficient number of iterations and all three stages are perfectly balanced.

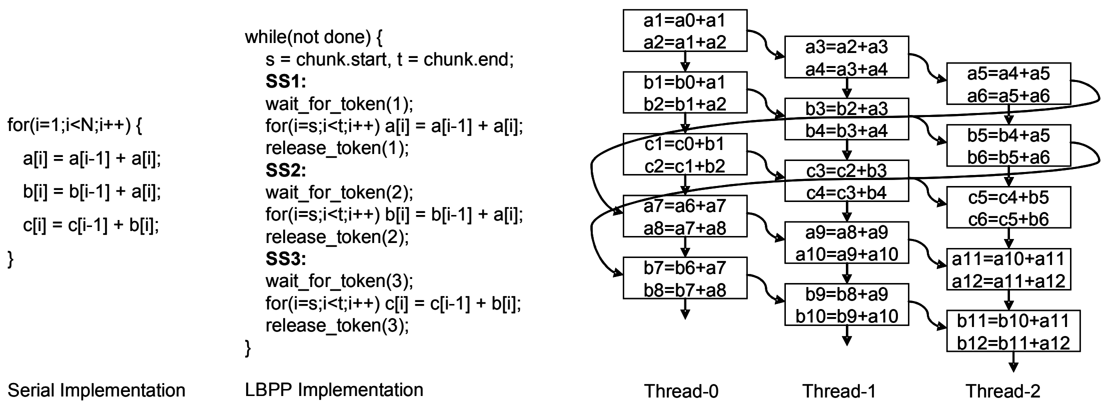
Figure 7.1(b) shows the availability of pipeline parallelism in an irregular loop. In this case, SS1 is a sequential stage that does the pointer chasing. It has cross-iteration dependency but no intra-iteration dependency. Stage PS1 updates each node pointed to by $p$ and depends only on SS1 computed in the same iteration. PS1 has no cross-iteration dependency and it is possible to compute different iterations concurrently for this stage, i.e. PS1 is a *parallel* stage. So, this irregular loop has a pipeline of one sequential and one parallel stage. We can use one thread for the sequential stage, and one or more for the parallel one to get a pipelined execution. Prior works address this form of execution as parallel-stage decoupled software pipelining (PS-DSWP) [ROR$^+$08]. Assuming we have enough cores, the maximum possible parallelism for this loop is determined by the relative weight of the sequential stage because we must use one thread for the sequential stage, but can use as many as necessary to make the parallel stage non-critical.

### 7.1.2 Load-balanced pipeline parallelism

A pipelined loop (regular or irregular) has one or more sequential stages, and zero or more parallel stages. It requires at least two stages for pipelining. The traditional way [RVVA04, ROR$^+$08] to implement such pipelining maps different stages to each core (or multiple cores, for parallel stages) and protects the intra-iteration dependency using synchronization. The cross-iteration dependency, on the other hand, is automatically satisfied by the program order. From now on, we use the term *traditional pipelining* or traditional pipeline parallelism (TPP) to refer to such execution.

Load-balanced pipeline parallelism executes all stages of a particular loop iteration in the same core, but distributes the iterations between cores in a manner more similar to conventional data parallel execution. First, it splits the loop iterations into a set of *chunks*, and consecutive chunks of the same stage are executed on different cores. LBPP executes all the stages of a chunk sequentially in a single core. It will execute several iterations (a chunk) of stage 1, followed by the same iterations of stage 2, etc. Once all the stages are finished, the thread starts executing the next assigned chunk, starting again at the first stage. LBPP maps the chunks to threads in round robin order.

This creates no correctness issue for the parallel stages, because they do



```
for(i=1;i<N;i++) {
   a[i] = a[i-1] + a[i];
   b[i] = b[i-1] + a[i];
   c[i] = c[i-1] + b[i];
}
```

```
while(not done) {
   s = chunk.start, t = chunk.end;
   SS1:
   wait_for_token(1);
   for(i=s;i<t;i++) a[i] = a[i-1] + a[i];
   release_token(1);
   SS2:
   wait_for_token(2);
   for(i=s;i<t;i++) b[i] = b[i-1] + a[i];
   release_token(2);
   SS3:
   wait_for_token(3);
   for(i=s;i<t;i++) c[i] = c[i-1] + b[i];
   release_token(3);
}
```

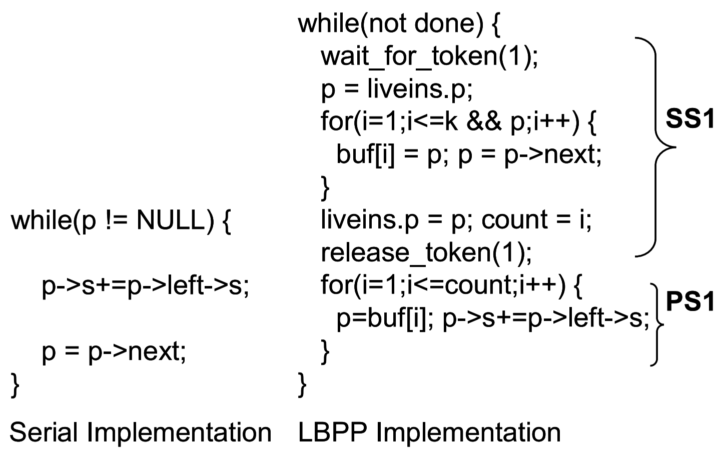
Figure 7.2: Implementation and execution of LBPP for a regular loop with a chunk size of 2 iterations that uses three threads.

not have any cross-iteration dependency, and the program order satisfies the intra-iteration dependency. However, there exist sequential stages with cross-iteration dependencies. LBPP handles that with a simple token-based synchronization mechanism and ensures properly ordered execution of the sequential stages.

LBPP uses a token for each sequential stage. Every participating thread waits for the corresponding token before executing a sequential stage. In that way, all of the iterations of a sequential stage execute in order – they execute in order on a single core within a chunk, and across chunks, the order is protected by synchronization. The parallel stages do not require tokens and LBPP executes them immediately. Once a sequential stage finishes execution, the current thread hands over the token to the next thread. Thus, the tokens for sequential stages move between threads in round robin order and guarantee serial execution of a sequential stage. LBPP uses spin locking to implement the tokens.

LBPP obtains concurrency in two ways – (1) between stages, as threads are typically working on different stages at different times, and (2) within parallel stages, as any thread can enter the parallel stage without regard to whether other threads are in the same stage.

Figure 7.2 shows the LBPP implementation of the regular loop in Figure 7.1(a). Here each chunk consists of two consecutive iterations. Unlike traditional pipelining, all three threads in this case execute the same function. In



```
                          while(not done) {
                            wait_for_token(1);
                            p = liveins.p;
                            for(i=1;i<=k && p;i++) {
                              buf[i] = p; p = p->next;
                            }
while(p != NULL) {          liveins.p = p; count = i;
                            release_token(1);
   p->s+=p->left->s;        for(i=1;i<=count;i++) {
                              p=buf[i]; p->s+=p->left->s;
   p = p->next;             }
}                         }
Serial Implementation     LBPP Implementation
```

Figure 7.3: Implementation of LBPP for an irregular loop with a chunk size of $k$ iterations.

traditional pipelining, we need to design separate functions for each pipeline stage, and then add synchronization.

We can also implement LBPP for the irregular loop in similar fashion. Figure 7.3 shows the LBPP implementation of the pointer chasing loop from Figure 7.1(b). The sequential stage, upon receiving the token, gets the start value of $p$ from the previous thread. Then, it does the pointer chasing for $k$ iterations and enqueues the pointers in a thread local buffer. After that, it forwards the current value of $p$ to the next thread and releases the token. The parallel stage dequeues the sequence of pointers from the local buffer and does the update operation. Thus, the local buffer transfers data from the sequential stage to the parallel stage and serves the same purpose as software queues do in traditional pipelining. Note that queues/local buffers may not always be necessary. The regular loop mentioned above is an example of that.

The performance of LBPP depends heavily on the *chunk size*. We define chunk size as the approximate memory footprint of a chunk that includes both the memory accesses done by the original loop itself, and the additional memory accesses to the local buffer. Chunk size is a critical tuning parameter in LBPP. It is highly correlated with the cost of synchronization and the data sharing overhead. Larger chunks better amortize the overhead of the synchronization (i.e., waiting for tokens) that is required for the sequential stages. The number of times that a thread needs to wait for the tokens is inversely proportional to the chunk size. Thus, when a sequential stage is critical (e.g., SS1 in Figure 7.1(b)), chunking minimizes the overhead of executing that stage in different cores.

Smaller chunks also create more coherence traffic when there exist data sharing between consecutive iterations. Most cross-iteration communication will stay on a core – only when that communication crosses chunk boundaries will it move between cores. Therefore, larger chunks reduce communication. In Figure 7.2, for example, we see both true and false sharing. Both iteration 5 and 6 use $a5$, $b5$, and $c5$. On the other hand, there exists false sharing between iteration 4 and 6 when any of $a4, a6$ or $b4, b6$ or $c4, c6$ occupy the same cache line. For a 64-byte cache line and assuming each of $a, b, c$ takes 8 bytes, we need to use at least 8 iterations per chunk to avoid excessive false sharing.

However, we cannot make chunks arbitrarily large for two reasons. First, smaller chunks exploit locality better. Chunks that do not fit in the private caches will evict the shared data between two stages and will cause unnecessary high-latency misses. Second, chunk size determines the amount of parallelism that can be extracted from a pipelined loop, especially when iteration counts are low. In Figure 7.2, for three threads, the chunk of 2 iterations gives parallelism of 2.25× whereas using 4 iterations per chunk will give parallelism of 1.8× (because our simple example has only 12 iterations). However, this gap diminishes when the loop iterates more. In the last example, for 10000 iterations, we get the same 3× parallelism for either of 2 or 4 iterations chunk. Even a chunk of 100 iterations gives 2.94× parallelism. Thus, from the parallelism point of view, the size of the chunk relative to the number of loop iterations is important.

LBPP provides three key advantages over traditional pipelining – locality, thread number independence, and load balancing.

**Locality**

LBPP provides better locality compared to traditional pipelining. In traditional pipelining, data shared between stages within an iteration always crosses cores. Since pipelined loops always share data between stages, this is the dominant communication. In LBPP, this communication never crosses cores. Traditional pipelining may avoid cross-core communication for loop-carried dependencies, but only if they go to the same stage. With LBPP, loop-carried dependencies only cross cores when they cross chunk boundaries, whether or not they are within a stage.

In Figure 7.1(a), the first two stages (SS1 and SS2) share $a$ while the last two stages share $b$. Similarly, in Figure 7.1(b), the parallel stage PS1 uses $p$ produced by SS1. Note that most of this sharing is read-write sharing, and all require cache-cache transfers with traditional pipelining. The communication cost was also identified as a key bottleneck for traditional pipelining on real systems [RVVA04, ROR$^+$08].

In contrast, LBPP executes all stages of the same chunk in the same core. All the intra-iteration communication between stages happens within the same core. In Figure 7.3, PS1 will find $p$ already available in the cache. The same thing happens for the regular loop in Figure 7.2. SS2 finds $a$ while SS3 finds $b$ in the private cache. LBPP can also exploit data sharing between distant stages, e.g. sharing between stage 1 and stage 4, etc.

For the example regular loop, assuming 3000 iterations, chunk size of 500, 8-byte values, and no false sharing, traditional pipelining would move 48,000 bytes for intra-iteration dependences while all loop carried communication is free. For LBPP, all intra-iteration communication is free, and we would move 144 bytes for loop-carried communication between chunks.

Chunking allows LBPP to target a certain level (either private or shared) of the cache hierarchy. By tuning the chunk size, LBPP can confine most memory operations to either the L1 cache or the L2 cache. In some sense, LBPP implements tiling between stages with the help of chunking. This, combined with the natural locality, allows LBPP to simply accommodate prefetching. We can simply add a stage to do prefetching, targeted the data that will be accessed by the next chunk, and that data stays in the cache until that stage executes. However, we do not implement that optimization in our results here.

**Thread number independence**

In Figure 7.2, there are three sequential stages. For traditional pipelining, the code created by the compiler would be very different depending on whether it expected two cores or three cores to be available for execution. In LBPP, the code is the same in either case. Thus, LBPP decouples the code generation from the expected thread count, simplifying compilation and providing better adaptation to runtime conditions. We can use any number of threads irrespective of the number of pipeline stages for a loop, even including more threads than stages. For one thread, the loop executes the three stages sequentially. For two threads, LBPP achieves pipelining and extracts as much as $2\times$ parallelism when all three stages are of equal weight (traditional pipelining will suffer from imbalance). The parallelism improves to $3\times$ when there are three threads. If we add more threads, the parallelism does not improve, but we might get additional speedup because of the data spreading (Chapter 3). Data spreading is a technique that distributes the data of a sequential loop over several private caches, creating the effect of a single large, fast private cache. We can emulate this mechanism just by growing the number of threads/cores until the working set fits in the caches, again without recompiling.

**Load balancing**

LBPP is inherently load balanced in the same way as traditional loop-level data parallelism, because each thread is doing the same work, only on different iterations of the loop. Thus, performance tends to scale linearly until we reach the maximum performance. That is not typically the case with traditional pipelining, although it will approach the same peak performance at the maximum number of threads.

Assume we have a pipeline of $k$ sequential stages with execution times $S_1, S_2 \ldots S_k$ and $SS_l$ is the largest stage with execution time, $S_{mx}$. We also assume that there are a large number of iterations and there is no synchronization overhead. Thus, if thread 1 releases a token at time $t$, thread 2 can grab that token and start executing at time $t$. Moreover, since there are enough iterations, we ignore the first and last few iterations where not all threads are active.

With these settings, there is an important observation. A thread will only wait for tokens when it needs to execute the bottleneck stage $SS_l$. That is because when it gets the token for $SS_l$, it would require the next token after $S_{mx}$ time and by that time, the next required token will be available. So, the average execution time of an iteration will be $T + W$ where T is $\sum_{i=1}^{k} S_i$, and $W$ is the waiting time for the token to execute $SS_l$.

We can compute $W$ for $n$ threads. A thread that just finishes $SS_l$ will require the token to execute this stage again after $T - S_{mx}$ time. On the other hand, the token will be available after $(n-1) * S_{mx}$ time. So, $W$ is $max(0, (n-1) * S_{mx} - (T - S_{mx}))$ or $max(0, n * S_{mx} - T)$. Thus, the average execution time of an iteration is $max(T/n, S_{mx})$ using $n$ threads. The equation also holds when there are one or more parallel stages. In that case, $T$ is $\sum_{i=1}^{k} S_i + \sum_{i=1}^{m} P_i$ where $P_1, P_2 \dots P_m$ are the execution times for the parallel stages. The expected parallelism of such a pipeline using LBPP for $n$ threads is given in the following equation:

$$LBPP_n = \frac{T}{max(T/n, S_{mx})} \tag{7.1}$$

Equation 7.1 shows that LBPP is load balanced and increases parallelism linearly until it reaches the theoretically maximum parallelism, i.e., $T/S_{mx}$. So, the minimum number of threads to gain the maximum parallelism is:

$$LBPP_{min} = ceil(T/S_{mx}) \tag{7.2}$$

Compared to that, traditional pipelining uses a single thread for each sequential stage, and one or more for each parallel stage. So, the minimum number of threads to achieve the maximum parallelism is:

$$TPP_{min} = k + \sum_{i=1}^{m} ceil(P_i/S_{mx}) \tag{7.3}$$

From Equation 7.2 and 7.3, $LBPP_{min}$ is always smaller than $TP_{min}$ unless all the stages have equal execution times. In that case, both values are the same. For example, assume the pipeline of a loop has 5 sequential stages with the execution time of 10, 15, 10, 20, and 5. Traditional pipelining will require 5 threads to reach the maximum parallelism of 3×. However, LBPP will require only 3 threads to reach that parallelism.

The load balancing of LBPP becomes more evident in the presence of parallel stages. Assume a pipeline has 2 sequential stages (execution times 10 and 10, respectively) and 2 parallel stages (40 and 40). If we have 4 cores, the maximum parallelism we can extract using traditional pipelining is $100/40 = 2.5\times$. However, LBPP extracts 4× parallelism and utilizes all threads 100%. LBPP provides more parallelism until we use 10 threads. In that case, both provide the maximum parallelism of 10×.

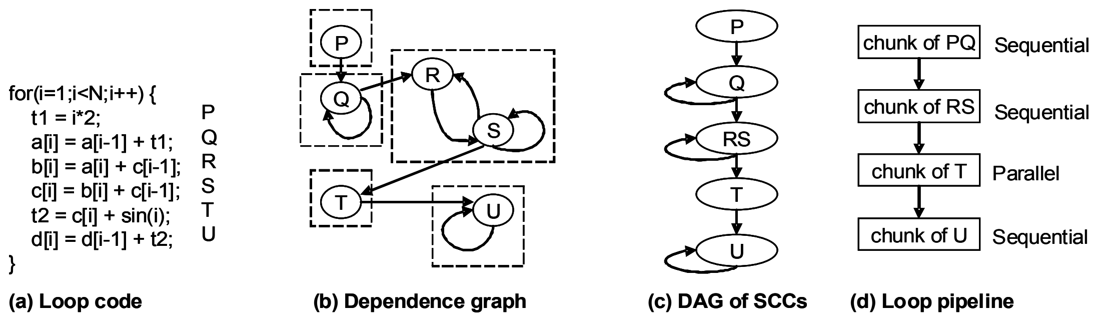
Equation 7.1 also suggests that for LBPP, the parallelism is independent of the number of stages. This gives enormous flexibility while designing the pipeline. Load balancing allows us to add more pipeline stages or redesign the current stages and still ensures the maximum parallelism as long as $T$ and $S_{mx}$ do not change.

If we account for synchronization cost, there is one case where traditional pipelining may have an advantage. Once we have enough threads that a sequential stage becomes the bottleneck, traditional pipelining executes that stage on a single core, while LBPP executes it across multiple cores. Both techniques will have some synchronization in the loop, but traditional pipelining may be able to avoid it when that stage only produces and no queuing is necessary (e.g., SS1 in our regular loop). However, for LBPP, it can be highly amortized with large chunks. In our experiments, this effect was, at worst, minimal, and at best, completely dominated by the locality advantages of LBPP.

## 7.2 LBPP Implementation

LBPP can be automatically implemented in the compiler and does not require any hardware support. Our implementation of LBPP follows very closely the implementation of traditional pipelining, at least in the initial steps. In fact, any traditionally pipelined loop can be automatically converted to take advantage of LBPP.

There are four steps to apply LBPP – DAG construction, pipeline design,



```
for(i=1;i<N;i++) {
    t1 = i*2;                  P
    a[i] = a[i-1] + t1;        Q
    b[i] = a[i] + c[i-1];      R
    c[i] = b[i] + c[i-1];      S
    t2 = c[i] + sin(i);        T
    d[i] = d[i-1] + t2;        U
}
```

Figure 7.4: Different steps of constructing the pipeline for a loop. Each rectangular box is a strongly connected component and cannot be pipelined.

adding synchronization, and chunking.

### 7.2.1 DAG construction

In the first step, we construct the dependence graph for a loop. For this step, we use the same methodology described in previous work [ORSA05, ROR$^+$08]. The dependence graph includes both data and control dependence. Figure 7.4(b) shows the dependence graph for the loop on the left side. An arc from node $x$ to node $y$ denotes that node $y$ must execute after the execution of $x$.

Next we identify the strongly connected components (SCC) in the dependence graph. This step is required because we cannot pipeline and execute the nodes in parallel when there exist cyclic dependencies between them. In Figure 7.4(b), node R and S are examples of having cyclic dependency. Executing all the nodes of an SCC in the same thread maintains the chain of dependency. The rectangular boxes in Figure 7.4(b) represent the SCCs. If we consider each SCC as a single node, we get a directed acyclic graph (DAG). This is called the *DAG of SCCs*. We can do a topological ordering for the DAG so that all edges are either self edges or go forward to subsequent nodes, i.e., there will be no backward edges. Figure 7.4(c) shows the DAG for this example loop.

In a DAG, the self arc represents cross-iteration dependency whereas forward arcs represent intra-iteration dependency. We can pipeline all the nodes of a DAG by making each node a different pipeline stage. This will satisfy all the dependencies. The nodes that do not have self arcs (P, T in Figure 7.4(c)) can be executed in parallel as well. These become the parallel stages in a pipeline. The others (Q, RS, U) become sequential stages.

### 7.2.2 Pipeline design

LBPP is load balanced and does not depend on how many threads will be used at runtime. This makes pipeline design easy and flexible.

From Section 7.1.2, we need to minimize both $T$ (the sum of all stage execution times) and $S_{mx}$ (the maximum of all stage execution times) for maximum parallelism. The DAG construction automatically minimizes $S_{mx}$. However, using all DAG nodes as separate stages may not minimize $T$. There are overheads for each additional stage. These include the synchronization overhead for sequential stages, decoupling overhead (loading and storing data from local buffers), reduced instruction level parallelism, and other software overheads. Chunking automatically amortize most of these overheads. To reduce them further, we can cluster stages as long as $S_{mx}$ does not increase.

We use a simple cost model to estimate the stage execution times. The model counts the number of different types of arithmetic operations, memory operations, and branches to assign weights to approximate the stage execution time. The model assumes a large weight for function calls or when the stage itself is a loop. Profiling is an option in this case to get better approximation. Note that for LBPP, we do not need a perfect partitioning of stages, but we should remove simple stages (easy to identify) that benefit less than the overheads induced. So, a precise execution time for complex stages (having function calls or loops) is not necessary.

As a first step of the clustering, we collapse simple producer stages with the corresponding consumer stages. In Figure 7.4(c), stage P computes $i * 2$ used by stage Q. Keeping P as a separate stage will cause a store operation (enqueue $i*2$) in P, and a load operation (deque $i*2$) in Q. Collapsing P with Q will remove these memory operations and the overheads for using one extra stage. This also compacts Q because the load operation is more expensive than computing $i * 2$

in most systems. As a rule of thumb, we collapse a stage unless it does enough computation that it is equivalent to doing at least three memory operations.

Next, we try to coalesce parallel stages similar to that described in the earlier work [ROR+08]. We combine two parallel stages PS1 and PS2 when there is no intervening sequential stage, SS such that there is an arc from PS1 to SS and SS to PS2 in the DAG of SCCs. We continue this process iteratively unless no more coalescing is possible. This process of coalescing does not change $S_{mx}$. So, from Equation 7.1, there is no negative impact on the level of parallelism.

LBPP works well if we design the pipeline using only the above two steps. Until this point, we do not need the value of $S_{mx}$ for the pipeline design. In the final step, we can further optimize the pipeline by collapsing the stages whose combined execution time does not exceed $S_{mx}$. Short parallel stages can also be combined with sequential stages (and be treated as sequential stages) in this process. A simple greedy algorithm is sufficient for this merging, because LBPP does not require the optimal solution. For example, assume a pipeline has 6 sequential stages of execution times 60, 40, 20, 30, 20, 10. In this case, both 60, 60, 60 and 60, 40, 50, 30 are equally good solutions.

The output of the pipeline design step is a sequence of sequential and parallel stages. Figure 7.4(d) shows the pipeline for our example loop. Next, we add necessary memory operations to transfer data from one stage to another stage. In our example, we buffer $t2$ in stage T and use that in stage U.

### 7.2.3 Chunking

Our compilation framework adds necessary code for chunking the pipeline stages. For a particular chunk size (passed as a runtime parameter), we compute the number of iterations per chunk (say $k$) using the memory footprint of a loop iteration. LBPP executes $k$ iterations of a stage before going to the next stage.

Figures 7.2 and 7.3 show the chunking of both regular and irregular loops. For regular loops (loop count is known), each thread independently identifies the chunks for themselves and execute those chunks. The threads exit when there are no more chunks to execute. For irregular loops, LBPP uses a sequential stage (SS1

```
while( node != root ) {
   while( node ) {
      if( node->orientation == UP )
        node->potential=node->basic_arc->cost+node->pred->potential;
      else {
        node->potential=node->pred->potential–node->basic_arc->cost;
        checksum++;
      }
      tmp = node;  node = node->child;
   }
   ................
   ................
}
```

Figure 7.5: Nested loop of *refresh_potential* function of *mcf.* The iteration count of the inner loop is not fixed.

in Figure 7.2) that catches the loop termination in one thread and forwards that information to other threads. LBPP automatically packs and unpacks necessary liveins for all the coordinations.

In this work, we statically compute the approximate memory footprint of a loop iteration. This is easy to do for loops that do not have inner loops. For nested loops, the iteration memory footprint of the outer loop depends on the number of iterations of the inner loop. Figure 7.5 shows the *refresh_potential* function of the Spec2006 benchmark *mcf.* In this case, the number of iterations of the inner loop is not fixed, so the memory footprint of the outer loop iteration changes dynamically. If we apply chunking to the outer loop and use a fixed value of $k$, chunks will not be of even size and load distribution between threads will be imbalanced. We solve this problem by adding a sequential stage that counts the number of inner loop iterations. Thus, we can compute a more accurate value of $k$ and do better load distribution – we terminate a chunk based on the inner loop count rather than the outer loop count. Note that in LBPP, adding a stage is inexpensive since it does not require a new thread.

In Section 7.4, we will examine other metrics besides memory footprint for determining chunk size.

Table 7.1: Microarchitectural details of the two systems that are used in our experiments.

| **CPU components** | **Intel Nehalem, AMD Phenom** |
|---|---|
| CPU Model | Core i7 920, Phenom II X6 1035T |
| Number of cores | 4, 6 |
| L1 cache size | 32-Kbyte, 64-Kbyte |
| L1 hit latency | 4 cycles, 3 cycles |
| L2 cache size | 256-Kbyte, 512-Kbyte |
| L2 hit latency | 10 cycles, 15 cycles |
| L3 cache size | 8-Mbyte shared inclusive, 6-Mbyte shared exclusive |
| L3 hit latency | 46 cycles, 45 cycles |
| DRAM hit latency | 190-200 cycles, 190-200 cycles |
| Cache to cache transfer latency | 42 cycles, 240 cycles |

### 7.2.4 Adding synchronization

In the final step, we add the synchronization operations (waiting for tokens and releasing tokens) to the sequential stages. The compilation framework creates the necessary tokens at the beginning of the program. At the start of a pipelined loop execution, the framework assigns all the corresponding tokens to the thread that executes the first chunk. When a thread completes a sequential stage, it passes the token to the next thread. There is also a barrier at the end of each pipelined loop to make all changes visible before proceeding to the computation following the loop.

## 7.3 Methodology

This section describes our methodology to evaluate LBPP. First, we describe the two systems used for our experiments and then describe the set of applications that we target to demonstrate LBPP.

### 7.3.1 CPU Systems

LBPP is expected to work well across all cache coherent systems. However, performance might vary because of the microarchitectural differences that exist between processors. These include the processor's memory hierarchy – geometry of the caches and latency to the different levels of cache or DRAM, cache coherence protocol, off-chip bandwidth, and the underlying interconnect. Thus, for our experiments, we choose two state of the art systems from different vendors. Table 7.1 summarizes the important microarchitectural information for the two systems.

The systems run Linux 2.6. We compile all our codes using GCC version 4.5.2 with "-O3" optimization level. We keep hardware prefetching enabled for all of the experiments. For power measurements, we use a "Watts up? .Net" power meter that measures the power at the wall and can report power consumption in 1-second interval with ±1.5% accuracy.

### 7.3.2 Applications

We apply LBPP on loops from a diverse set of applications chosen from different benchmark suites – Spec2000, Spec2006, Olden, SciMark2. We also pick two important irregular applications (also used in previous work) – *ks* (Kernighan-Lin graph partitioning algorithm), and *otter* (an automated theorem prover).

Pipeline parallelism is most interesting when there are at least two stages in the pipeline. This excludes the loops that are completely data parallel (only one parallel stage), and the sequential loops that have a single SCC in the DAG of SCCs. Thus, we exclude several benchmarks from these suites whose key loops do not have multiple pipeline stages. In addition, our compilation framework currently only handles C code, further limiting candidate benchmarks.

Table 7.2 shows the loops of interest, corresponding function names, and the description of the benchmarks. We select a loop if it is not parallel and contributes at least 10% to the total execution time, when executed serially with the reference input. Table 7.2 also shows the type of each loop, its contribution to the total execution time, and the pipeline structure identified by the compiler. All the irregular loops involve pointer chasing. Most of these loops have inner loops.

Table 7.2: List of benchmarks explored in our experiments. Here, *ss* represents sequential stage where *ps* stands for parallel stage.

| **Function name** | **Bench-mark name** | **Suite** | **Loop type** | **Contrib. Phenom, Nehalem** | **Pipeline structure** |
|---|---|---|---|---|---|
| match, train_match | art | Spec2000 | Regular | 77%, 63% | ss1 ps1 |
| f_nonbon | ammp | Spec2000 | Regular | 12%, 11% | ss1 ps1 ss2 |
| smvp | equake | Spec2000 | Regular | 68%, 51% | ss1 ps1 ss2 |
| SetupFastFull PelSearch | h264ref | Spec2006 | Regular | 30%, 29% | ss1 ss2 ps1 |
| P7Viterbi | hmmer | Spec2006 | Regular | 99%, 99% | ps1 ss1 |
| LBM_perform StreamCollide | lbm | Spec2006 | Regular | 99%, 99% | ps1 ss1 |
| refresh_potential | mcf | Spec2006 | Irregular | 19%, 25% | ss1 ps1 ss2 |
| primal_bea_mpp | mcf | Spec2006 | Regular | 61%, 46% | ps1 ss1 |
| FindMaxGp AndSwap | ks | Graph partitioning | Irregular | 100%,100% | ss1 ps1 ss2 |
| BlueRule | mst | Olden | Irregular | 77%, 70% | ss1 ps1 ss2 |
| find_lightest_ geo_child | otter | Theorem proving | Irregular | 10%, 3% | ss1 ps1 ss2 |
| SOR_execute | ssor | SciMark2 | Regular | 99%, 99% | ps1 ss1 |

## 7.4 Results

We evaluate LBPP in several steps. First, we use a set of simple microbenchmarks to explore the potential and different characteristics of LBPP. Then, we evaluate its impact on the real applications described in Section 7.3. Finally, we study LBPP's impact on the power and energy consumption.

For all of our experiments, we also implement traditional pipelining (TPP). We do our best to implement it as efficiently as possible. This is straightforward because the compiler-extracted pipeline structure and even the optimization of the stages (coalescing, etc.) are the same for both techniques. In fact, both LBPP and traditional pipelining use similar types of synchronization.

```
for(i=1;i<N;i++) {
  SS1:
  a[i] = sin(a[i-1]+a[i]+1);
  SS2:
  b[i] = sin(b[i-1]+a[i]+1);
  SS3:
  c[i] = sin(c[i-1]+b[i]+1);
  SS4:
  d[i] = sin(d[i-1]+c[i]+1);
  SS5:
  e[i] = sin(e[i-1]+d[i]+1);
}
```
(a) **mb_load**

```
for(i=2;i<N;i++) {
  SS1:
  a[i] = (a[i-2] + a[i-1] + a[i])/3.0;
  PS1:
  b[i] = sin(a[i])*cos(i);
  SS2:
  c[i] = (c[i-1] + a[i]   + b[i])/3.0;
  PS2:
  d[i] = sin(c[i]) + M_PI;
}
```
(b) **mb_ubal**

```
t = start_node;
while(t != stop_node) {
    SS1:
    tmp = t;
    t = t->next;
    SS2:
    s += tmp->left->a;
}
```
(c) **mb_local**

```
for(i=1;i<N;i++) {
  SS1:
  a[i] = a[i] + f(a[i-1]);
  SS2:
  b[i] = a[i] + f(b[i-1]);
  SS3:
  c[i] = b[i] + f(c[i-1]);
  SS4:
  d[i] = c[i] + f(d[i-1]);
}
```
(d) **mb_dyn**

Figure 7.6: Source code of the microbenchmarks that explore different aspects of LBPP.

### 7.4.1 Microbenchmarks

We design several microbenchmarks that are simple to understand and whose pipeline structures are easy to recognize. These microbenchmarks capture a variety of characteristics including the amount of data transfer between stages, the ratio of sequential and parallel stages, the relative weight of stages, and the dynamic nature of the pipeline. Using microbenchmarks allows us to adapt the code to focus on a particular characteristic.

Figure 7.6 shows the source code of the microbenchmarks that we use. The stages and the type of stages are also marked in the figure. All of these microbenchmarks run many iterations and have large working sets unless specified otherwise. The speedup numbers given in this section and afterwards are all normalized to the single thread execution with no pipelining.

**Load balancing and number of stages** Our first microbenchmark, *mb_load* has five sequential stages and all of them have similar execution times. Thus, the pipeline is mostly balanced. Consecutive stages have some data sharing, e.g., SS2 and SS3 share *b*. However, the amount of data sharing compared to the computation per stage is small, because the *sin* function is expensive.

We apply LBPP and TPP on this microbenchmark for different number of threads. For LBPP, we use the same pipeline structure for all thread combinations. However, the 5-stage pipeline given above is only applicable for five threads when we use TPP. Thus, we make the best possible partition of the stages into 2, 3, and

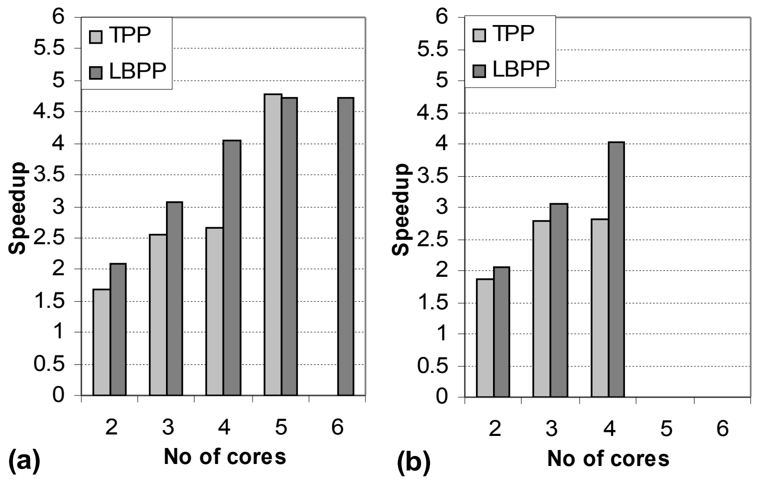


Figure 7.7: Scalability of *mb_load* – (a) Phenom (b) Nehalem. The pipeline has five stages of equal weight. LBPP provides better load balancing until both techniques reach the maximum speedup.

4 groups to run with 2, 3, and 4 threads, respectively, but a balanced partitioning is not possible. Figure 7.7 shows the speedup using both techniques on the two experimental systems. We use several chunk sizes for both LBPP and TPP and show the data for the best chunk size. Chunking lets TPP use synchronization and enqueuing once for a group of iterations, like LBPP.

LBPP improves performance linearly in both systems. The maximum possible parallelism for this loop is 5×. LBPP provides 4.7× speedup for five cores in the AMD system and 4× for four cores in the Intel system. From Equation 7.1, the parallelism will not increase once it gets limited by the largest sequential stage. We see the same trend here. For the AMD machine, the speedup stays the same beyond five cores.

Figure 7.7 also demonstrates the importance of decoupling the pipeline design and the number of threads to be used. Traditional pipelining gives the same speedup as LBPP for five cores. However, it loses as much as 35% (4.1× vs. 2.7×) and 30% (4× vs. 2.8×) for four threads in the AMD and Intel system, respectively, despite using the optimal partitioning of the stages. This happens due to the lack of load balancing. With four partitions of the five stages, the largest partition is

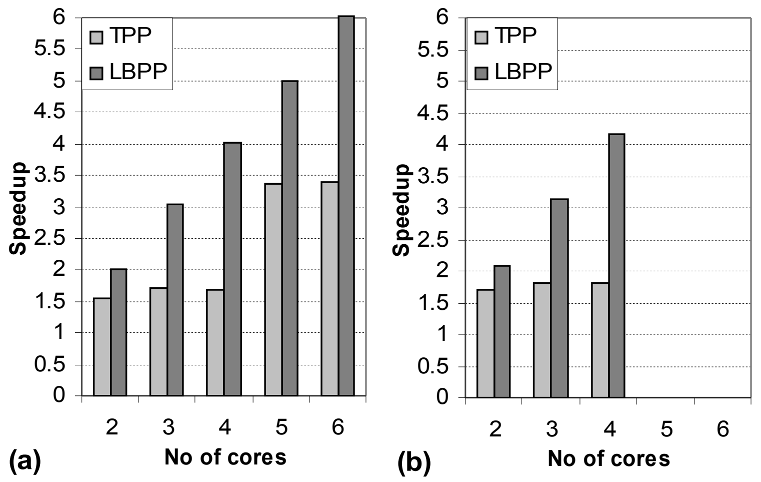


Figure 7.8: Scalability of *mb_ubal* – (a) Phenom (b) Nehalem. The pipeline has a mix of parallel and sequential stages of different weights. LBPP provides as large as 2× performance improvements over TPP.

twice as big as the smallest. LBPP also outperforms traditional pipelining while using 2 or 3 cores.

**Unbalanced stages** LBPP effectively handles the pipeline where there are multiple parallel stages and the stages are not balanced. The microbenchmark *mb_ubal* has two parallel stages and two sequential stages of relative weight of 60, 30, 5, and 5, respectively. Thus, the sequential stages are not dominant here.

Figure 7.8 shows the performance for this pipelined loop. For TPP, we use the best possible partitioning when there are less than four threads. If we have more than four threads, we assign the extra threads to the parallel stages.

LBPP provides linear performance improvements for all thread combinations in both machines. The sequential stages here are much shorter than the parallel stages. LBPP does not bind stages to threads and dynamically assigns stages to threads. With LBPP, it is possible that all threads are executing the largest parallel stage PS1 at a time, where TPP only gets stage parallelism when it can explicitly assign multiple threads to a stage. LBPP provides 6× speedup for six cores in the AMD machine and 4.2× for four cores in the Intel machine.

In this case, traditional pipelining suffers from load imbalance since the

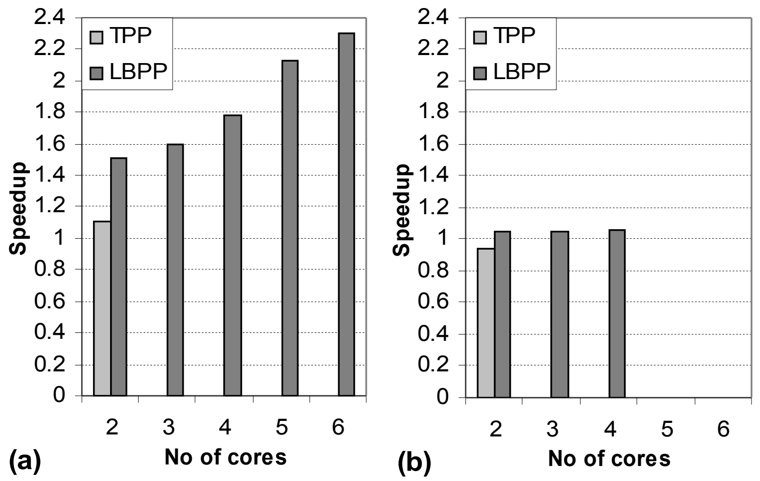


Figure 7.9: Scalability of *mb_local* – (a) Phenom (b) Nehalem. The irregular loop has two sequential stages. LBPP exploits locality in both systems. In the Phenom, LBPP also shows data spreading.

two small sequential stages and sometimes the smaller parallel stage, PS2, occupy cores and sit idle waiting for the data from the largest stage, PS1. This results in performance loss for all core combinations. The loss increases with the imbalance. For four cores (each stage gets its own core), the loss is more than 55% in Nehalem, and 58% in Phenom. The situation does not improve much even when we use extra threads for PS1 and PS2. The performance loss is 44% for six cores in the AMD machine.

**Locality** Our third microbenchmark *mb_local* explores the locality issue in pipelining. For this microbenchmark, we use a 1 Mbyte working set and execute the loop multiple times. Thus, the same data is reused again and again. The loop is an irregular loop with two sequential stages. SS1 does the pointer chasing and supplies the set of pointers to SS2, which does the accumulation. Thus, there exists significant data sharing between the two stages relative to the amount of computation per iteration.

Figure 7.9 compares the performance between LBPP and TPP. LBPP outperforms TPP by 36% and 11% for Phenom and Nehalem, respectively, using two cores (TPP cannot use more than two cores). In both cases, two cores are enough to exploit the available parallelism. However, the speedup is less than 2× because of the imbalance in the pipeline, and also for the negative impact of decoupling, i.e., reduced instruction level parallelism. The imbalance is much higher in Nehalem.

TPP does not do well in either machine, because it requires cache to cache transfers to send the pointers from SS1 to SS2. In LBPP, the pointers stay in the local cache. We measure the number of references to the last level cache using hardware performance counters. TPP causes 16% more references than that of LBPP in Nehalem.

The performance impact of losing locality is more prominent in Phenom than it is in Nehalem. This directly correlates with the latency of cache to cache transfers. The latency is much higher in Phenom (Table 7.1), making locality more critical.

Figure 7.9 also explains LBPP's ability to exploiting data spreading. Nehalem does not experience this, because the working set does not fit in the combined L2 cache space. For the AMD machine (with more cores and larger L2 caches), the working set gets distributed and fits in the aggregate L2 cache space as we add more cores. This does not increase parallelism, but reduces average memory access time. The effect improves LBPP's performance up to 2.3× while using six cores, even though there are only two sequential stages.

**Varying stages** Our final microbenchmark, *mb_dyn* captures the case when the stage execution times change over different iterations. The function *f* picks a random number between 0 to 500 based on the argument and does some counting operations. Thus, the relative weight of the stages vary over time.

Figure 7.10 shows the scalability for this workload. There are four sequential stages and LBPP gives linear speedup up to four cores in both machines, extracting the maximum parallelism. Traditional pipelining works equally well in Nehalem given enough cores, but falls short in the Phenom. For four cores, the loss is around 7%. We observe the effect of an unbalanced pipeline (four stages cannot be perfectly partitioned into three groups) when we use three cores for traditional pipelining. LBPP outperforms TPP by 61% in Phenom, and by 49% in Nehalem.

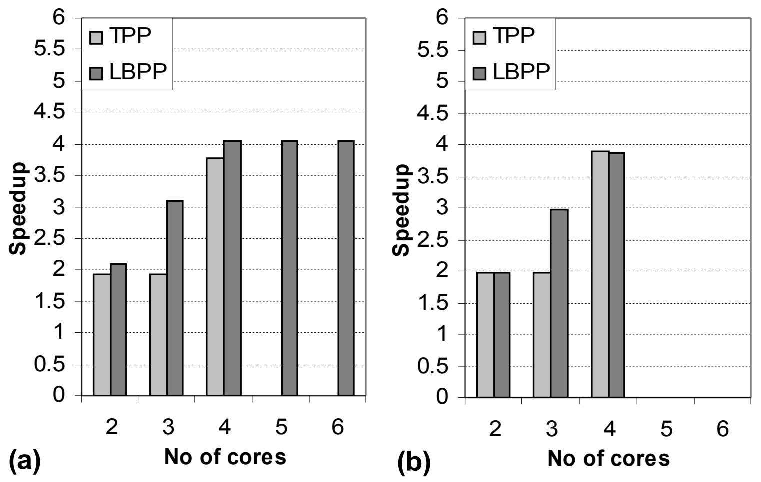


Figure 7.10: Scalability of *mb_dyn* – (a) Phenom (b) Nehalem. The execution time of all four sequential stages vary during different iterations. LBPP handles these variations better.

### 7.4.2 Impact of chunk size

Chunking is a critical component for LBPP, because it amortizes the overhead due to data sharing and synchronization. In this section, we analyze the performance impact for different chunk sizes on LBPP. So far, we have used chunking based on the approximate memory footprint of an iteration. We can also do chunking using the number of iterations directly, or using the execution time of an iteration (using expected iteration latency to compute a chunk iteration count). Thus, chunk size can be a simple iteration count or execution time (in $\mu s$) other than the memory footprint.

Figure 7.11 shows the performance for our four microbenchmarks using the three chunking techniques. We give the results for the AMD Phenom machine and use two cores. We vary the chunk size from 1 to 10000 iterations, $0.1\mu s$ to $100\mu s$, and 1KB to 512KB for the iteration based, execution based, and footprint based chunking, respectively.

Iteration based chunking is the simplest. Figure 7.11(a) explains the need for chunking. Without chunking (chunk size of one iteration in the graph), the performance seriously suffers. The loss can be as high as 93% compared to the sin-

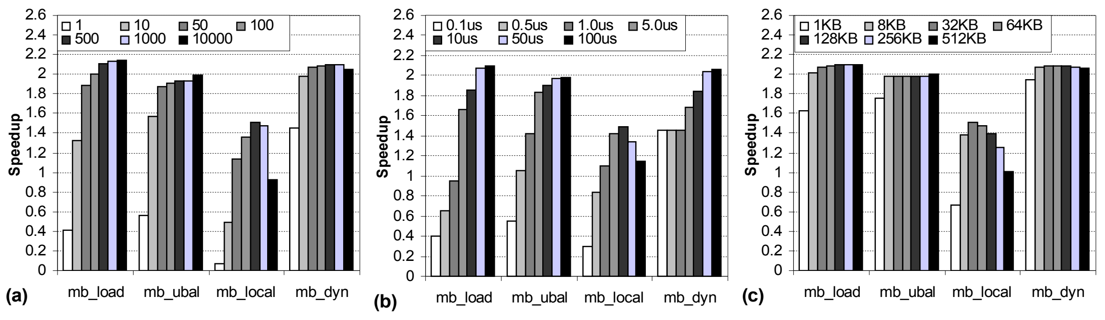


Figure 7.11: Performance impact in LBPP of different chunk sizes in the AMD Phenom system – (a) Iteration based, (b) Execution time based, (c) Memory footprint based. Chunks must be large enough to amortize the synchronization overhead.

gle threaded execution that uses no pipelining. Only *mb_dyn* shows some speedup, because it does more computation per iteration than the others. Using 10 iterations per chunk is also not sufficient. The tightest loop, *mb_local* cannot amortize the synchronization overhead and shows negative performance. We see significant performance improvements for 50 iterations and it improves further for 500 iterations across all microbenchmarks. Thus, LBPP requires significant computation per chunk to keep the synchronization overhead under control.

Figure 7.11(b) explains the correlation between the chunk execution time and the synchronization overhead. LBPP cannot amortize the overhead when the chunks execute for $0.5\mu s$ or less. Here, *mb_dyn* is an outlier, because each of its iteration takes $2.5\mu s$ on average and we use at least one iteration per chunk. Out of the four microbenchmarks, *mb_load* has four sequential stages and it performs more synchronization per iteration than the others do. We observe the impact in the graph. Overall, LBPP amortizes the overhead well and reaches closes to $2\times$ speedup when the chunk size is at least $10\mu s$.

Memory footprint based chunking handles the locality issue. We can tune the chunk size to target a particular level of the cache hierarchy. Memory footprint also strongly correlates with the execution time. Figure 7.11(c) shows that chunk sizes starting from 8KB work well across our microbenchmarks. It also shows that by keeping the chunk size limited to 32KB or 64KB, we can confine the data transfers from SS1 to SS2 for *mb_local* in the L1 cache and get maximum benefit.

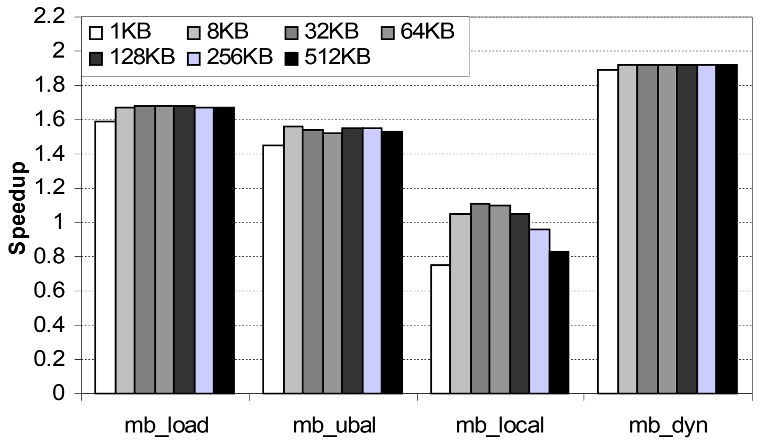


Figure 7.12: Impact of chunking on Traditional pipelining. Similar to LBPP, chunks smaller than 8K do not work well.

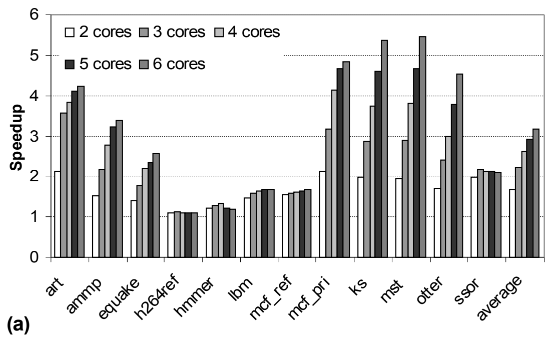
Traditional pipelining shows sensitivity to the chunk size, too. Figure 7.12 shows the impact of using different memory footprint based chunk sizes. In this case, the primary advantage is to reduce the instances of synchronization.

### 7.4.3 Application Performance

This section describes the performance of LBPP on real applications. Table 7.2 gives detailed information about the loops and their corresponding pipeline structure. We select two loops from two different functions for *mcf*. From Table 7.2, all the loops have at least one parallel stage and one sequential stage. Thus, the mix of parallel and sequential stages is very common.

Figure 7.13 shows the loop level speedup of LBPP for different number of cores in the AMD Phenom machine. We normalize the performance using both single thread execution (left graph), and using TPP with the same number of cores (right graph). For TPP, when there are more cores than the number of stages, we apply the extra cores to the parallel stage. We try with the seven footprint based chunk sizes given in Figure 7.11(c) and show the result for the best chunk size.

LBPP provides significant performance improvements across all loops, an average speedup of 1.67× for just two cores. The pointer chasing loop in the



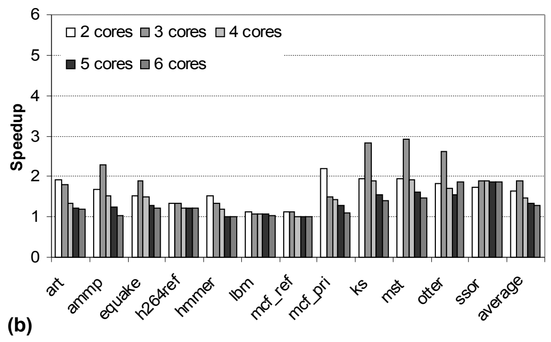


Figure 7.13: Performance of LBPP on sequential loops from real applications using different core counts in AMD Phenom – (a) normalized to single thread, (b) normalized to TPP.

complex integer benchmark, *mcf* shows 1.57× speedup justifying the importance of loop level pipelining. Some of the loops (e.g., *ks*, *mst*, etc.) show linear scalability. The speedup can be as high as 5.5× in some cases. On average, the performance improves from 1.67× to 3.2× when we apply six cores instead of two cores. Some of the loops do not scale well, because the parallel stage is not large enough and the sequential stages start to dominate according to Equation 7.1. The loops from *equake* and *lbm* have larger parallel stages, but lose some scalability due to the data sharing and off chip bandwidth limitation, respectively. The *equake* loop has significant write sharing across different iterations, enough that the communication across chunk boundaries makes an impact. The *lbm* loop is bandwidth limited and cannot exploit additional computing resources.

LBPP outperforms traditional pipelining in almost all cases. The difference is much bigger for smaller core counts, because of the load balancing and locality issue. LBPP outperforms TPP by 65% for two cores, and by 88% for three cores on average. Traditional pipelining closes the gap for higher core counts (5 to 6 cores). For six cores, LBPP wins by 27%. In that case, there are enough cores for all stages and load balancing does not remain a critical issue. This reduces the locality issue to some extent, especially when the parallel stage gets data from another stage. TPP still moves large amounts of data, but with more cores, it is better able to hide the communication. We see this effect in the *refresh_potential*

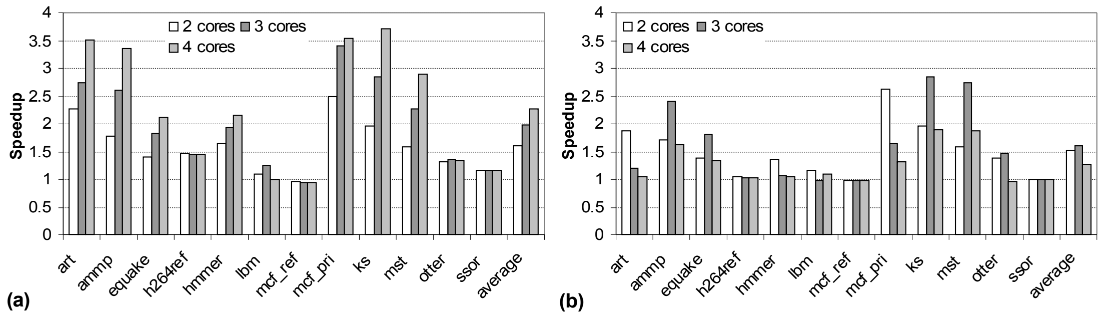


Figure 7.14: LBPP performance on the Intel Nehalem for real application loops – (a) normalized to single thread, (b) normalized to TPP.

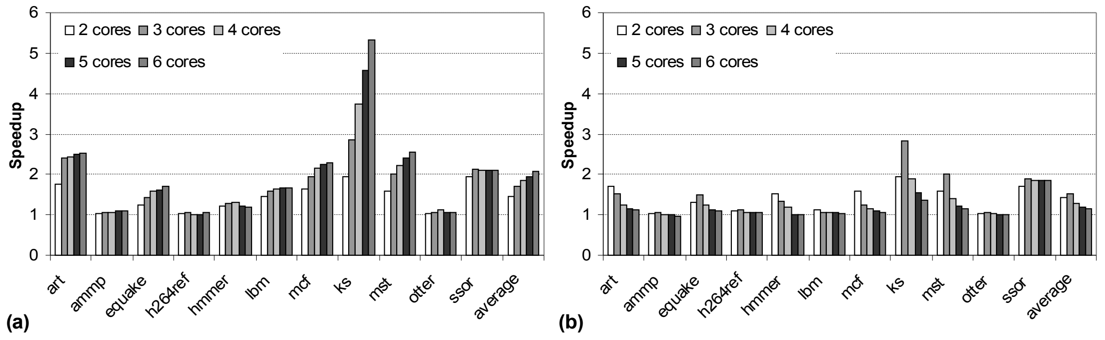


Figure 7.15: Application level speedup in AMD Phenom – (a) normalized to single thread, (b) normalized to TPP.

loop of *mcf*, for example.

We also do the same set of experiments in our Nehalem system (Figure 7.14). LBPP provides 1.6× and 2.3× speedup on average using two and four cores, respectively. This is 50% and 26% higher than what traditional pipelining offers. Thus, LBPP performs well across different architectures.

The results so far show the performance of the individual loops. Figure 7.15 describes the impact on the application level for the AMD machine, since those loops only represent a fraction of total execution time. The application performance varies depending on the loop's contribution to the total execution time, but also on how the loop interacts with the rest of the program. For example, the serial part might suffer more coherence activity because of the data spread-

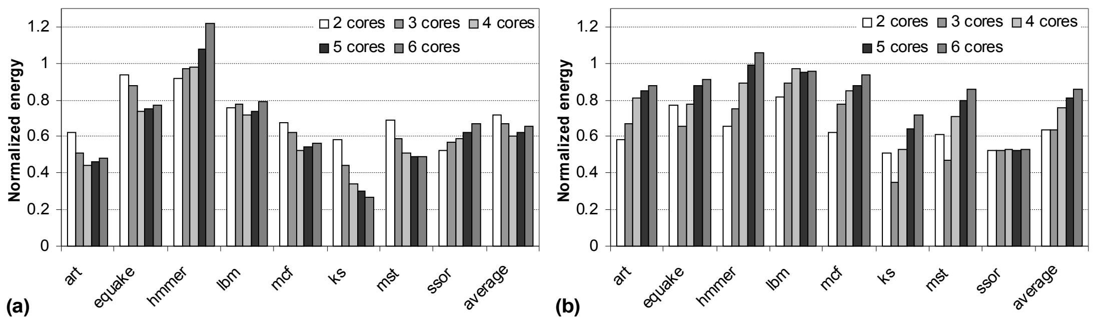


Figure 7.16: Energy consumption for the selected benchmarks in Phenom – (a) normalized to single thread, (b) normalized to TPP. Here, lower is better.

ing by the pipelined loop. Overall, even at the application level LBPP speedups are high, including outperforming TPP by 42% and 51% for two and three cores, respectively.

### 7.4.4 Energy Considerations

We measure the total system power using a power meter to understand the power and energy tradeoff for LBPP. The meter reports the power consumption in 1-second intervals, prohibiting power measurement at the loop level, because most of the loops execute for much shorter than 1 second. Thus, we pick the benchmarks where the selected loops contribute at least 60% to the total run time and dominate the power consumption.

Figure 7.16 shows the normalized energy consumption in the AMD systems for different numbers of cores. LBPP provides significant energy savings over single thread execution across all core counts, due to decreasing execution times. LBPP beats TPP for all core counts. The energy savings is 36% on average for two and three cores. The improvement in locality makes a significant difference. This can be seen at large core counts where the difference in energy is far higher than the difference in performance. In some cases, we even observe LBPP consuming less *power* than that of TPP even though it executes faster. For *mst* and *equake*, the power savings is around 8% and 7%, respectively, while using three cores.

## 7.5 Conclusion

This chapter describes Load-Balanced Pipeline Parallelism. LBPP is a compiler technique that takes advantage of the pipelined nature of sequential computation, allowing the computation to proceed in parallel. Unlike prior techniques, LBPP preserves locality, is naturally load-balanced, and allows compilation without a priori knowledge of the number of threads. LBPP provides linear speedup on a number of important loops when prior techniques fail to do so.

LBPP works by chunking, or executing a number of iterations of a single stage, before moving onto the next stage. For a sequential stage, a synchronization token is sent to another thread to continue with the next chunk. In this way, intra-iteration communication is always local, and even cross-iteration communication is minimized. Also, because all threads execute all stages, it is naturally load-balanced.

LBPP outperforms prior pipeline parallel solutions by up to 50% or more on full applications, especially for lower thread counts. It provides even more striking energy gains, by reducing both runtimes and decreasing expensive cache-to-cache transfers.

## Acknowledgments

Portions of this research were funded by the US National Science Foundation grants CCF-1018356 and NSF-0643880 and by SRC Grant 2086.001.

This chapter contains material from “Load-balanced Pipeline Parallelism”, by Md Kamruzzaman, Steven Swanson and Dean M. Tullsen, which has been submitted for possible publication in PLDI’13: Proceedings of the 2013 ACM SIGPLAN Conference on Programming Language Design and Implementation. The dissertation author was the primary investigator and author of this paper.

# Chapter 8

# Related Work

This dissertation addresses the parallelization wall problem and proposes several techniques that extract more performance, and energy efficiency from state of the art parallel hardware. The techniques proposed here improve application performance by accelerating each single thread using multiple processing cores. This dissertation also focuses on improving scalability of parallel applications by combining different parallelization techniques and on improving energy efficiency by dynamic frequency scaling.

In this chapter, we place our techniques in the context of previous approaches in the related area. We broadly categorize the prior approaches in three main sections – improving single thread execution, improving parallel execution, and improving energy efficiency. Next, we describe the relevant research work in each section and compare the main goals, technical differences, and performances with our techniques.

## 8.1 Improving single thread execution

Accelerating single thread execution has been a key area of interest for long time. There have been both hardware and software approaches. Some of the works apply the basic techniques that we use in our techniques including helper threads, prefetching, resource aggregation, decoupled execution, etc.

### 8.1.1 Resource aggregation

Data spreading aggregates the cache resources of different processors. Both inter-core prefetching and load-balanced pipeline parallelism can also achieve the advantage of data spreading. There have been several prior works that focus on the same theme of resource aggregation (primarily cache resources) from different processing units.

Pierre Michaud [Mic04] proposes a multicore design that works in the similar spirit of data spreading. In this system, the hardware monitors the cache misses, does some affinity analysis, and performs thread migration to take the advantage of aggregate cache space. Compared to that, data spreading is a software only technique and works on state of the art real systems. So, it does not incur the cost of additional hardware when the technique is not effective, or not utilized. Data spreading also applies across sockets and can aggregate last level caches. We find that most of the benefit comes in that way, because it amortizes the thread migration cost easily. Most importantly, the compiler driven analysis done in data spreading spreads the working set lot more effectively.

Chakraborty, et al., [Kou06] present a technique called Computation Spreading, which also tries to leverage other caches via migration like what data spreading does. They migrate threads so that some cores execute exclusively operating system code, and others execute exclusively user code. In this way, they get separation of data that is not typically shared over short time frames, and co-location of data more likely to be shared. However, they do not achieve the effect of working set spreading that is the primary contributor to the speedups of data spreading.

Cooperative Caching [CS06] is an architectural technique with the same goal as data spreading – using caches from neighboring cores to support the execution of a single thread. They do this by allowing caches to store data that have been evicted from other private caches. Thus, caches that are lightly used or idle can act as large victim caches [Jou90]. However, this can only transform misses into cache-to-cache transfers. Both data spreading and inter-core prefetching effectively transform misses into local hits.

Other work [HKS$^+$05, M. 05, LSK04] has suggested blurring the distinction between private and shared caches even further to improve cache locality. Core fusion processors [IKKM07] take a different approach and allow multiple cores to be dynamically combined into a single larger core. Conjoined core designs [KJT04] allow two cores to share even the lowest-level (L1) caches. These approaches require significant changes to the hardware and do not involve thread migration for sharing resources. So, none of these hardware techniques work across multiple chips.

### 8.1.2 Helper thread prefetching

Prefetching is an important technique to speed up memory-intensive applications. There has been significant work on both hardware prefetchers [Jou90, CB95] and software controlled prefetchers [MLG92, APD01, CH02]. The introduction of multithreaded and multicore architectures introduced new opportunities for prefetchers. Multithreaded architectures are a natural target, because threads share the entire cache hierarchy.

Helper thread prefetchers [CSK+99, CWT+01, IBR03, ZS01, Luk01, KY02, KLW+04, LWW+02, SPR00, CTWS01, LDH+05] accelerate computation by executing prefetch code in another hardware context, or core. The idea first came in the context of simultaneous multithreaded processor [TEL95] and later got extended to multicore architectures. In these techniques, helper threads predict future load addresses by doing some computation and prefetch the corresponding data to the nearest level of shared cache. Prior work focus on different aspects of helper thread prefetching including the target systems, architectural design, runtime implementation, how the helper threads are generated, etc.

Chappell, et al. [CSK+99] first propose the idea of simultaneous subordinate microthreading that use microthreads (i.e., helper threads) in a multithreaded processor to implement a variety of optimizations including prefetching data in advance to assist the primary computing thread. In their system, hardware manages the spawning of microthreads, and a separate hardware structure called microRAM stores the microthread code. The system uses hand built microthread code.

The work in [CWT+01] introduces a helper thread prefetching technique that targets the inorder Itanium $^{\text{TM}}$processor. The authors find that the number of loads that causes the bulk of cache misses and pipeline stalls is very small. They classify these loads as delinquent loads and use helper threads to prefetch the corresponding data. In their design, the system spawns a helper thread well ahead of the execution (the trigger point) of the delinquent load instruction. The helper thread executes the set of critical instructions (known as p-slice) required to compute the load address and prefetch data. In this case, helper threads are much short-lived. Zilles and Sohi propose a similar system [ZS01] for out of order processors. In addition to prefetching, their work use helper threads to improve branch prediction accuracy. Both of these two systems use hand constructed helper thread code.

Collins, et al. [CTWS01] extend their prior work on helper thread prefetching and show that it is possible to construct the p-slice (helper thread code) automatically in hardware. Their system keeps track of the delinquent load and goes backwards through the traces of committed instructions to construct the p-slice that computes the load address.

In the work on software controlled pre-execution [Luk01], the author uses the original program instead of the reduced version to get the advantage of helper thread prefetching. In this approach, helper threads run the target section of the original program in the pre-execution mode that ignores the exceptions and does not commit any store operations. To ensure latency tolerance, the system uses multiple helper threads to prefetch multiple data streams in parallel.

Liao, et al. [LWW$^+$02] propose a compiler implementation of constructing the p-slice for delinquent loads. Like prior approaches, they use profile guided analysis to identify the delinquent loads, and then use a secondary pass to the compiled binary to construct the p-slices. The system also identifies proper spawning points for helper threads and adds necessary synchronization instructions to automate the entire process. Binaries constructed this way show promising performance (around 87%) for inorder processor, but not so much (around 5%) for out of order processor.

The works so far only exploit the indirect advantages (prefetching, improvements in branch prediction accuracy). Roth and Sohi come up with the idea of data-driven multithreading [RS01], where the main thread gets the prefetching advantage as well as reuses some of the computational results produced by the helper thread. In this case, both threads share the same physical register file, and the hardware adapts register renaming to deliver some of the values produced by the helper thread to the main thread. A profile driven analysis extracts helper threads from program traces automatically.

Prior research has also targeted prefetching across cores [SPR00, IBR03, LDH$^+$05]. In Slipstream processors [IBR03], a reduced version of the program speculatively runs ahead of the main program in another core. Multiple specialized hardware pipes transport information (including loaded values) from one to the other. The trailing main execution gets prefetching advantage and also bypasses some of the computation when the outcomes of the speculative computations are correct. The hardware generates the reduced version by detecting the instructions that can be skipped based on the past runtime behavior. The follow-on work [IBR03] shows the potential of this technique for an effective self-invalidation scheme to reduce cache coherence activities. Slipstream processors also target reliability.

Lu, et al. [LDH$^+$05] demonstrate helper thread prefetching on a real CMP system that has a shared cache. They use a dual core UltraSPARC IV+ with shared L2 cache and implement a dynamic optimization system that executes in a separate core. It monitors the main program execution to select program regions that have delinquent loads, generates the helper thread code, and runs that code in the same core. Brown, et al. [BWC$^+$01] propose changes to CMP hardware and coherence to enable a thread on one core to effectively prefetch for a thread on a separate core.

The construction of helper threads, or p-slice is an important component for helper thread prefetching. There are several ways to generate helper threads including by hand [CWT$^+$01, ZS01], in hardware [CTWS01, SPR00], profile driven compiler analysis [RS01, LWW$^+$02], etc. Zhang, et al. [ZTC07] and Lu, et al. [LDH$^+$05] describe the generation of helper threads in dynamic compilation systems. Kim and Yeung [KY02] propose the first source-level compiler to automatically generate the helper thread code. Their compiler first uses profile information to identify the cache-missing memory references and the enclosing loops. After that, the compiler constructs a program dependence graph and performs data-flow, and control-flow analysis to extract the non-critical computations from the target loop.

In another work by Kim, et al. [KLW$^+$04], the authors implement a helper thread prefetching system entirely in software to target the hyperthreaded (i.e., simultaneous multithreaded) Intel Pentium 4 processor. They find that in that hyperthreaded system, the overhead due to synchronization, thread management, and thread conflicts dominates the performance advantages.

Gummaraju, et al. [GR05] implement a compilation framework that uses helper thread prefetching to map a program written for stream processors to general purpose processors. In their system, there are three threads – one for execution, one for data prefetching, and one that manages execution and prefetching. They target SMT threads.

Prescient instruction prefetch [ACH$^+$04] is another helper thread-based technique that improves performance by prefetching instructions instead of data. None of our techniques target reducing instruction cache misses. However, data spreading can be adapted to spread the instructions as well.

One of our techniques described in this dissertation, inter-core prefetching also belongs to the genre of helper thread prefetching. However, inter-core prefetching has some unique characteristics that make it a lot more powerful technique than prior approaches. First, it allows cross-core prefetching into private caches with no new hardware support, and it does not require a shared cache. So, we can apply ICP on any cache coherent parallel hardware including multi-socket systems. Second, it uses thread migration to use the prefetched data locally. This combines the best of shared cache prefetching and SMT prefetching. Third, the helper threads in ICP prefetch a chunk of data to amortize the synchronization and thread management overhead. We can tune the number of helper threads and chunk size to make ICP effective on different types of applications. Finally, unlike all prior approaches, main thread may not be always executing in ICP. For very memory intensive applications, ICP may temporarily use all cores to run helper threads. This ensures better resource utilization.

### 8.1.3 Runahead execution

Runahead execution is a hardware based latency tolerant technique. Unlike helper thread prefetching, runahead execution [DM97, MSWP03] does not require extra hardware contexts or cores. During the long latency load operation, instead of blocking, it assumes a dummy value and switches to the runahead mode to continue execution. The runahead mode is very similar to the pre-execution mode described above and does not cause any change in the program state. When the original load operation is satisfied, the application switches back to the normal execution. This improves memory level parallelism but still places all of the burden of prefetching, demand misses, and execution on a single core.

### 8.1.4 Decoupled architecture

Inter-core prefetching uses separate threads to access memory and perform computation. This is similar in spirit to decoupled access/execute architectures [Smi82]. These architectures use a queue to pass data from a memory thread to a compute thread. The queue requires that the two threads be tightly coupled, and that the memory thread correctly computes the result of all branches. As a result, there is rarely enough "slack" between the threads to allow the memory thread to issue memory requests far enough in advance to fully hide their latency. The Explicitly-Decoupled architecture [GH08] uses the queue only for passing branch outcomes and shares the entire cache hierarchy to get the advantage of prefetching.

Inter-core prefetching accomplishes the benefit of decoupled access/execute architecture without specialized hardware, and thus delivers a long sought solution for this elegant execution model. Comparing to the hardware implementation, the private cache of a helper core serves as the "queue", but because inter-core prefetching places no ordering constraints on the accesses, it can completely decouple the threads.

### 8.1.5 Speculative multithreading

Speculative multithreading [MGT98, QMS$^+$05, KT99, SBV95] straddles the line between traditional and non-traditional parallelism. Most of this work strives to maintain single-thread software semantics, while the hardware executes threads in parallel.

In multiscalar processors [SBV95], the control flow graph of the program gets divided into several tasks that execute in parallel in different processing units. The hardware manages the inter-task data communication and ensures the correctness of the speculative execution by adopting the recovery mechanism whenever required. Multiscalar uses compiler support to identify the tasks. Marcuello, et al. [MGT98] propose another speculative multithreaded architecture that dynamically extracts multiple threads of control in hardware from a sequential program. Their system primarily targets loops and execute different iterations of the same loop speculatively. Krishnan, et al. [KT99] propose a speculative system that targets the CMP architecture. They use a binary annotation tool to generate the speculative threads. Mitosis [QMS$^+$05] is a hardware-software approach that extends the concept further by adding helper threads to precompute critical data.

All of these speculative multithreaded processors use large amount of additional hardware to manage speculation. So, there is a lot of underutilized hardware when parallel applications or multiprogrammed workloads execute. This is a big concern, because most of the systems are nowadays power constrained. All the techniques in this dissertation are non-speculative and software only. These techniques do not have to worry about correctness or recovery scheme. In addition, there is no penalty when we do not use any of these techniques. Speculative techniques are normally orthogonal to non-speculative techniques. For example, we can speculatively parallelize the main thread execution in inter-core prefetching. We can also use speculation to compute load addresses that cannot be predicted.

### 8.1.6 Decoupled software pipelining

Decoupled software pipelining [RVVA04, ORSA05] is a non-speculative technique that exploits pipeline parallelism on the loop level. Using dependence analysis, it automatically partitions an iteration of the loop into several stages and executes them in different processing units in a pipelined fashion to extract parallelism. Raman, et al. [ROR$^+$08] propose parallel-stage DSWP that identifies some stages as parallel and executes them in a data parallel manner.

Our load-balanced pipeline parallelism shares the same concept of separating the loop iteration into a set of sequential stages or a mix of sequential and parallel stages, but the execution model is different. Each processing core does not execute a particular pipeline stage in LBPP. It might be the case that all cores are executing the same pipeline stage at a time. Unlike decoupled software pipelining, which assumes hardware support for efficient synchronization and core to core communications, LBPP uses chunking to amortize synchronization overhead and naturally exploits locality to reduce inter-core communication. DSWP deals parallel stages specially, but this is not the case in LBPP. Here, parallel stages just do not wait for the tokens.

Vachharajani, et al. [VRR$^+$07] adds speculation on top of decoupled software pipelining. They speculate on few dependencies to extract more pipeline stages and use a special commit thread for the recovery process. Spice [RVhRA08] uses value prediction of the loop live-ins to enable speculative threading. Huang, et al. [HRJ$^+$10] combine speculative parallelization with parallel stage DSWP. Load-balanced pipeline parallelism is non-speculative and orthogonal to the speculative techniques. We can speculatively parallelize the sequential stages that dominate the execution. We can also use speculation to create additional pipeline stages to extract more parallelism.

### 8.1.7 Others

Event-driven compilation systems [ZCT05, ZCT06, ZTC07] use idle cores to invoke a compiler at run-time to perform optimizations in response to events detected by hardware performance monitors. DeVuyst, et al. [DTK11] use the dynamic optimization framework for runtime parallelization of single-threaded legacy programs. They use the support of hardware transactional memory for efficient parallel processing. Both data spreading, and inter-core prefetching use profile guided analysis and perform simple loop transformations based on that information. An event-driven compiler could do this type of analysis runtime and then recompile to apply these optimizations transparently.

Cache blocking [MC69, LRW91] is a compilation technique that reorders computation on a single core to increase locality and reduce cache misses. Our techniques could actually be complementary to techniques such as this, because they can reduce the reuse distance for only a subset of the accesses and must still incur some capacity misses on any structure that does not fit in the cache.

## 8.2 Improving parallel execution

Improving the scalability of parallel applications is another area of interest. There have been works on new programming models, extracting parallelism, load-balancing, synchronization overhead, etc.

**Traditional parallelism** Countless researchers have studied techniques for creating parallel code, either by discovering it automatically in the compiler [DRV00], providing primitives and libraries for the programmer [CJP07, Mue93, Phe08], providing parallelized libraries for key computations [Int07], or building parallelism into the programming language [Che93].

In this dissertation, we provide techniques that work on each parallel thread. We mainly experiment with the workloads that are either sequential, or express parallelism by using Pthreads [Mue93] and OpenMP [CJP07]. However, our techniques apply to most forms of traditional parallelism and can improve nearly all of them, even including mechanisms such as MPI [SOW$^+$95], that do not communicate through shared memory.

**Pipeline parallelism** Pipeline parallelism [NATC09, BL12, GMV08, LTST11] in another powerful parallel programming model that is well studied. Navarro, et al. [NATC09] gives an analytical model for pipeline parallelism using queuing theory to estimate the performance. Bienia, et al. [BL12] use PCA analysis to characterize pipeline applications from data parallel applications. Communications between different pipeline stages is shown to be an important component. The cache-optimized lock-free queue [GMV08] is a software construct that reduces communication overhead. Lee, et al. [LTST11] propose hardware solutions to the queuing overhead. Chen, et al. [CYH10] improves decoupled software pipelining using a lock-free queue.

Thies, et al. [TCA07] provides a mechanism to assist programmers to extract coarse-grained pipeline parallelism in C programs. They primarily target streaming applications.

Load-balanced pipeline parallelism achieves the effect of pipeline parallelism implicitly. However, it does not always execute in pipelined fashion. At some point, all cores may execute the same pipeline stage like a data parallel application does. In LBPP, all core to core communications occur through shared memory. So, it does not warrant for additional hardware based fast communication mechanisms.

**DOACROSS parallelism** DOACROSS [Cyt86] is another technique that distributes loop iterations like LBPP to extract parallelism, but it does not support arbitrary control flow inside the loop body and does not distinguish between sequential and parallel stages. This distinction is important, because a lot of serial codes have pipelines composed of dominant parallel stages. The execution model of LBPP extracts maximum possible parallelism for any number of cores. It is also easy for LBPP to load-balance and accelerate irregular pointer chasing codes.

There are also works on constructing efficient scheduling of doacross parallelism. Chen and Yew [CY94] focus on dependence graph partitioning to reduce synchronization overhead for DOACROSS loops. LBPP can use their algorithm as well to construct the pipeline.

Loop distribution [KM90] is a related technique that splits a loop into several loops by isolating the parts that are parallel (have no cross iteration dependencies) from the parts that are not parallel and can leverage data parallelism. However, it does not pipeline sequential stages with the parallel stages like LBPP does. In addition, it might require unlimited queuing space when there are data transfers between different parts of the loop iteration.

**Load balancing** Load balancing among threads has also received attention in the past. Work stealing [BL99] is a common scheduling technique where cores that are out of work steal threads from other cores. Feedback-driven threading [SQP08] shows, analytically, how to dynamically control the number of threads to improve performance and power consumption. It dynamically measures data synchronization and bus bandwidth usage to modify its threading strategy. Suleman, et al. [SQKP10] propose a dynamic approach to load balancing for pipeline parallelism. Their technique tries to find the limiter stage at runtime and allocates more cores to it. They assume at least one core per stage. Sanchez, et al. [SLY$^+$11] use task stealing with per-stage queues and a queue backpressure mechanism to enable dynamic load balancing for pipeline parallelism.

LBPP naturally achieves load-balancing. In more complex cases, when the weight of a pipeline stage varies dynamically, LBPP still does a fair job because of chunking. For even better load balancing, we can add a lightweight sequential stage to measure the load of a particular stage dynamically and partition the work according to that. Adding lightweight sequential stages is close to free in LBPP.

**Others** Researchers have also improved scalability of parallel code by combining different types of traditional parallelism and by dynamically changing thread behavior. Researchers have proposed hybrid models [LC08] that combine subsets of Pthreads, OpenMP, and MPI as well as systems that combine multiple forms of parallelism. For example, Gordon, et al. [GTA06] provide a compiler system that finds the right combination of task parallelism, data parallelism, and pipeline parallelism for stream programs. All of our techniques target programs written for general purpose processors.

## 8.3 Improving energy efficiency

Energy efficiency is a key concern nowadays for all computing platforms from smartphones to datacenters. There have been works on hardware to make power efficient processors as well as on software to reduce the operating power or to improve the energy efficiency.

Weiser, et al. [WWDS94] take an operating system stand for processor power reduction. They analyze the traces of multiple applications that are scheduled to run and monitor the system idle time that they use to adjust the processor speed. They find that scheduling algorithms have the potential to reduce processor power consumption while satisfying the important deadlines.

Hsu, et al. [HK03] propose a compiler algorithm that exploits dynamic voltage scaling (DVS) to reduce application power consumption. Their implementation uses instrumented code to measure the impact of using different voltage/frequency settings for a particular region and then use that information to construct an application specific DVS schedule. Xie, et al. [XMM03] demonstrate an analytical model that a compiler can use for DVS scheduling.

Choi, et al. [CSP04] implement a runtime system that uses an embedded performance monitoring unit (PMU) to identify memory intensive regions and apply DVFS on that. The PMU computes the CPU clock cycles required to execute instructions and the number of external memory access clock cycles to determine the memory intensiveness. Poellabauer [Poe05] takes a similar approach that monitors data cache miss rates using hardware performance counters and uses that as a feedback. The work by Magklis, et al. [MSS$^+$03] target a multiple clock domain architecture and use profile guided analysis to determine the voltage settings of those domains dynamically. Hotta, et al. [HSK$^+$06] use DVFS algorithm in a cluster to slow down the nodes that wait for data from another nodes. They also use profiling to identify the proper regions of code.

The work in [KGyWB08] by Kim, et al. describe the design of on-chip voltage regulators to provide fast per-core DVFS. They show that per core DVFS allows the opportunity of applying higher frequency to a compute intensive thread while applying lower frequency to memory bound thread. This optimizes energy efficiency rather than using a fixed voltage/frequency setting for each chip. Fast per-core DVFS also helps when same thread has both CPU bound and memory bound regions.

In contrast to these approaches, underclocked software prefetching proposed in this dissertation does not require complex DVFS algorithm. The decoupled execution model automatically provides memory intensive regions (i.e., prefetching part) where we can apply dynamic frequency scaling. We also do not need to worry about performance degradation or the right frequency settings. We can increase the number of helper threads to take care of that.

Thread motion [RWB09] by Rangan, et al. takes a different approach to save power. It uses different voltage/frequency settings for different cores, and the thread moves around across different cores depending on the computing need. So, the technique trades the switching latency of DVFS for the cost of thread migration and loosing the working set. Underclocked software prefetching also uses thread migration, but finds the working set ready in the new core.

Jeong, et al. [JKK$^+$12] propose the design of a low overhead power gating technique that power gates an active core when it stalls during long latency memory operations. Underclocked software prefetching and inter-core prefetching will also get benefit from such architectures. We can apply power gating when helper threads wait for the main thread and also, during the prefetch operations.

## Acknowledgments

Portions of this research were funded by the US National Science Foundation grants CCF-1018356 and NSF-0643880 and by SRC Grant 2086.001.

This chapter contains material from "Software Data Spreading: Leveraging Distributed Caches to Improve Single Thread Performance", by Md Kamruzzaman, Steven Swanson and Dean M. Tullsen, which appears in PLDI'10: Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation. The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2010 by the Association for Computing Machinery, Inc. (ACM).

This chapter contains material from "Inter-core Prefetching for Multicore Processors Using Migrating Helper Threads", by Md Kamruzzaman, Steven Swanson and Dean M. Tullsen, which appears in ASPLOS'11: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Lan-

guages and Operating Systems. The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2011 by the Association for Computing Machinery, Inc. (ACM).

This chapter contains material from “Coalition Threading: Combining Traditional and Non-Traditional Parallelism to Maximize Scalability”, by Md Kamruzzaman, Steven Swanson and Dean M. Tullsen, which appears in PACT’12: Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, September 2012. The dissertation author was the primary investigator and author of this paper. The material in this chapter is copyright ©2012 by the Association for Computing Machinery, Inc. (ACM).

# Chapter 9

# Summary

It is a recurring theme in computer architecture that hardware parallelism always precedes and typically exceeds software parallelism – forcing software to struggle to keep pace. We saw this with vector machines, VLIW, and wide superscalar processors. We are seeing it today, or will shortly, to a greater extent than ever before with the pervasive parallelism brought about by multithreaded and multicore processors. The scenario known as the parallelization wall is the key bottleneck against the performance growth that we used to see in the past.

This dissertation characterizes the parallelization wall problem and introduces several techniques as a potential solution. We find that the key to solving the parallelization wall is to accelerate the single thread execution using multiple cores and thus to bridge the gap between software and hardware parallelism. This will improve serial applications as well as parallel applications, because all applications will eventually experience scalability limitations.

The non-traditional techniques presented in this dissertation are software only and employ multiple cores to speedup the single thread execution. We find that it is not always necessary to extract thread level parallelism to leverage multiple cores. In software data spreading, we intelligently spread the working set across different private caches and essentially construct a larger private cache to improve locality. We develop an automated compiler algorithm to identify the loops that can take such advantages. Data spreading provides up to 3.3× speedup for some applications.

We devote a greater part of this dissertation behind the inter-core prefetching technique and its extensions. Inter-core prefetching implements the first helper thread prefetching technique that allows remote prefetching (in a different core) but local accessing of data. Like data spreading, it also uses thread migration as a key primitive. We find inter-core prefetching to be very effective on real systems providing up to 63% performance on average. More importantly, inter-core prefetching sometimes provides more speedup than what traditional parallelization gives making it very attractive for parallel applications as well.

Coalition threading intelligently combines traditional and non-traditional parallelism to improve scalability of parallel applications. We show the effective integration of inter-core prefetching and data parallelism for several parallel applications. This implies that using all cores for computation may not be always optimal. We develop a compiler algorithm that identifies those cases very accurately.

Underclocked software prefetching leverages the decoupled execution model of inter-core prefetching for saving power. It applies dynamic frequency scaling on helper threads that are memory intensive and less sensitive to frequency. We show that using low frequency helper threads provides better energy efficiency than using fewer high frequency helper threads while still maintains the same level of performance. In some sense, this demonstrates a way of applying power to the right component.

Finally, this dissertation proposes load-balanced pipeline parallelism, a technique that decomposes loop iterations into a set of pipeline stages, but execute them in a data parallel fashion. LBPP does not handle parallel stages specially, yet ensures maximum possible parallelism for any number of cores. We evaluate LBPP in the context of two state of the art systems and find that it provides linear speedup to a lot of serial loops.

In conclusion, this dissertation shows promising directions to handle the parallelization wall problem and brings attention to several interesting points in this regard. We can extract significant speedup to a lot of single thread executions without parallelizing it. Thread migration can serve as a powerful optimization primitive to improve performance on multicore systems. Decoupling memory accesses provides another dimension of parallelism that we can exploit for performance as well as for power. Software techniques allow more options to achieve locality. And, coalition threading opens up a new way of expressing parallelism.

# Bibliography

[ACH$^+$04] Tor M. Aamodt, Paul Chow, Per Hammarlund, Hong Wang, and John P. Shen. Hardware support for prescient instruction prefetch. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, 2004.

[ADE$^+$01] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. Specomp: A new benchmark suite for measuring parallel computer performance. In *Workshop on OpenMP Applications and Tools*, 2001.

[APD01] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Data prefetching by dependence graph precomputation. In *Proceedings of the 28th annual international symposium on Computer architecture*, 2001.

[BBDS93] Davd H. Bailey, Eric Barzcz, Leonardo Dagum, and Horst D. Simon. NAS parallel benchmark results. *IEEE Concurrency*, February 1993.

[BGK96] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd annual international symposium on Computer architecture*, 1996.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[BL99] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5), September 1999.

[BL12] Christian Bienia and Kai Li. Characteristics of workloads using the pipeline programming model. In *Proceedings of the 2010 international conference on Computer Architecture*, 2012.

[BT08] Jeffery A. Brown and Dean M. Tullsen. The Shared-Thread Multiprocessor. In *International Conference on Supercomputing*, June 2008.

[BWC$^+$01] Jeffery A. Brown, Hong Wang, George Chrysos, Perry H. Wang, and John P. Shen. Speculative precomputation on chip multiprocessors. In *In Proceedings of the 6th Workshop on Multithreaded Execution, Architecture, and Compilation*, 2001.

[CB95] Tien-Fu Chen and Jean-Loup Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, (5), May 1995.

[CH02] Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002.

[Che93] DY Cheng. A survey of parallel programming languages and tools. 1993.

[CJP07] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[CS06] Jichuan Chang and Gurindar S. Sohi. Cooperative caching for chip multiprocessors. In *Proceedings of the 33rd annual International Symposium on Computer Architecture*, June 2006.

[CSK$^+$99] R. Chappell, J. Stark, S. Kim, S. Reinhardt, and Y.N. Patt. Simultaneous subordinate microthreading (ssmt). In *Proceedings of the international symposium on Computer Architecture*, May 1999.

[CSP04] Kihwan Choi, Ramakrishna Soma, and Massoud Pedram. Dynamic voltage and frequency scaling based on workload decomposition. In *Proceedings of the International Symposium on Low power electronics and design*, 2004.

[CTWS01] J.D. Collins, D.M. Tullsen, H. Wang, and J.P. Shen. Dynamic speculative precompuation. In *Proceedings of the International Symposium on Microarchitecture*, December 2001.

[CWT$^+$01] J.D. Collins, H. Wang, D.M. Tullsen, C.J. Hughes, Y.-F. Lee, D. Lavery, and J.P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *Proceedings of the International Symposium on Computer Architecture*, July 2001.

[CY94] Ding-Kai Chen and Pen-Chung Yew. Statement re-ordering for doacross loops. In *Proceedings of the ICPP - Volume 02*, 1994.

[CYH10] Wen Ren Chen, Wuu Yang, and Wei Chung Hsu. A lock-free cache-friendly software queue buffer for decoupled software pipelining. In *Computer Symposium (ICS), 2010 International*, 2010.

[Cyt86] Ron Cytron. Doacross: Beyond vectorization for multiprocessors. In *ICPP'86*, 1986.

[Dev] E. E. Devices. Watts up? .net, https://www.wattsupmeters.com.

[DM97] James Dundas and Trevor Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 11th international conference on Supercomputing*, 1997.

[DRV00] Alain Darte, Yves Robert, and Frederic Vivien. *Scheduling and Automatic Parallelization.* Birkhauser Boston, 1st edition, 2000.

[DTK11] Matthew DeVuyst, Dean M. Tullsen, and Seon Wook Kim. Runtime parallelization of legacy code on a transactional memory system. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, 2011.

[EBSA$^+$11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, 2011.

[FAWR$^+$11] Michael Floyd, Malcolm Allen-Ware, Karthick Rajamani, Bishop Brock, Charles Lefurgy, Alan J. Drake, Lorena Pesantez, Tilman Gloekler, Jose A. Tierno, Pradip Bose, and Alper Buyuktosunoglu. Introducing the adaptive energy management features of the power7 chip. *IEEE Micro*, March 2011.

[FCH$^+$08] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. Liblinear: A library for large linear classification. *J. Mach. Learn. Res.*, 9:1871–1874, June 2008.

[GH08] Alok Garg and Michael C. Huang. A performance-correctness explicitly-decoupled architecture. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, 2008.

[GMV08] John Giacomoni, Tipp Moseley, and Manish Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proceedings of the PPoPP*, 2008.

[GR05] Jayanth Gummaraju and Mendel Rosenblum. Stream programming on general-purpose processors. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, 2005.

[GTA06] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2006.

[Hen00] John L. Henning. SPEC CPU2000: Measuring cpu performance in the new millennium. *Computer*, July 2000.

[Hen06] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, September 2006.

[HK03] Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003.

[HKS$^+$05] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S.W. Keckler. A NUCA substrate for flexible CMP cache sharing. In *International Conference on Supercomputing*, June 2005.

[HMN09] Daniel Hackenberg, Daniel Molka, and Wolfgang E. Nagel. Comparing cache architectures and coherence protocols on x86-64 multicore smp systems. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[HRJ$^+$10] Jialu Huang, Arun Raman, Thomas B. Jablin, Yun Zhang, Tzu-Han Hung, and David I. August. Decoupled software pipelining creates parallelization opportunities. In *Proceedings of the CGO*, 2010.

[HSK$^+$06] Yoshihiko Hotta, Mitsuhisa Sato, Hideaki Kimura, Satoshi Matsuoka, Taisuke Boku, and Daisuke Takahashi. Profile-based optimization of power performance by using dynamic voltage scaling on a pc cluster. In *Proceedings of the 20th international conference on Parallel and distributed processing*, 2006.

[IBR03] Khaled Z. Ibrahim, Gregory T. Byrd, and Eric Rotenberg. Slipstream execution mode for cmp-based multiprocessors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003.

[IKKM07] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: accommodating software diversity in chip multiprocessors. *SIGARCH Comput. Archit. News*, May 2007.

[Int07] MKL Intel. Intel math kernel library, 2007.

[JKK$^+$12] Kwangok Jeong, A.B. Kahng, SeokHyeong Kang, T.S. Rosing, and R. Strong. Mapg: Memory access power gating. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, 2012.

[Jou90] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the international symposium on Computer Architecture*, June 1990.

[KBW12] Wonyoung Kim, David Brooks, and Gu-Yeon Wei. A fully-integrated 3-level dc-dc converter for nanosecond-scale dvfs. *IEEE Journal of Solid-State Circuits*, January 2012.

[KGyWB08] Wonyoung Kim, Meeta S. Gupta, Gu yeon Wei, and David Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *International Symposium on High-Performance Computer Architecture*, 2008.

[KJT04] Rakesh Kumar, Norman P. Jouppi, and Dean M. Tullsen. Conjoined-core chip multiprocessing. In *Proceedings of the International Symposium on Microarchitecture*, December 2004.

[KLW$^+$04] D. Kim, S. Liao, P. Wang, J. Cuvillo, X. Tian, X. Zou, H. Wang, D. Yeung, M. Girkar, and J. Shen. Physical experiment with prefetching helper threads on Intel's hyper-threaded processors. In *International Symposium on Code Generation and Optimization*, March 2004.

[KM90] Ken Kennedy and Kathryn S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of the Supercomputing*, 1990.

[Kou06] Koushik Chakraborty and Philip M. Wells and Gurindar S. Sohi. Computation spreading: Employing hardware migration to specialize CMP cores on-the-fly. In *Proceedings of the international conference on Architectural support for programming languages and operating systems*, November 2006.

[KT99] Venkata Krishnan and Josep Torrellas. A chip-multiprocessor architecture with speculative multithreading". *IEEE Transactions on Computers*, September 1999.

[KTR$^+$04] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-isa heterogeneous multicore architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 64. IEEE Computer Society, 2004.

[KY02] D. Kim and D. Yeung. Design and evaluation of compiler algorithm for pre-execution. In *Proceedings of the international conference on Architectural support for programming languages and operating systems*, October 2002.

[LC08] Ewing Lusk and Anthony Chan. Early experiments with the openmp/mpi hybrid programming model. In *Proceedings of the International Conference on OpenMP in a new era of parallelism*, 2008.

[LCM$^+$05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 conference on Programming Language Design and Implementation*, June 2005.

[LDH$^+$05] Jiwei Lu, Abhinav Das, Wei-Chung Hsu, Khoa Nguyen, and Santosh G. Abraham. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, 2005.

[LRW91] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The cache performance and optimization of blocked algorithms. In *Proceedings of the international conference on Architectural support for programming languages and operating systems*, April 1991.

[LSK04] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for cmps. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, February 2004.

[LTST11] Sanghoon Lee, Devesh Tiwari, Yan Solihin, and James Tuck. Haqu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor. In *HPCA'11*, 2011.

[Luk01] Chi-Keung Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th International Symposium on Computer architecture*, July 2001.

[LWW$^+$02] S.S.W. Liao, P.H. Wang, H. Wang, G. Hoflehner, D. Lavery, and J.P. Shen. Post-pass binary adaptation for software-based speculative precomputation. In *Proceedings of the conference on Programming Language Design and Implementation*, October 2002.

[M. 05] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *Proceedings of the 32nd annual International Symposium on Computer Architecture*, June 2005.

[MC69] A.C. McKeller and E.G. Coffman. The organization of matrices and matrix operations in a paged multiprogramming environment. *Communications of the ACM*, March 1969.

[McG06] Harlan McGhan. Niagara 2 opens the floodgates. *Microprocessor Reports*, November 2006.

[MGT98] Pedro Marcuello, Antonio González, and Jordi Tubella. Speculative multithreaded processors. In *12th International Conference on Supercomputing*, November 1998.

[Mic04] Pierre Michaud. Exploiting the cache capacity of a single-chip multicore processor with execution migration. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture*, February 2004.

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, 1992.

[MSS$^+$03] Grigorios Magklis, Michael L. Scott, Greg Semeraro, David H. Albonesi, and Steven Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *Proceedings of the 30th annual international symposium on Computer architecture*, 2003.

[MSWP03] Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, 2003.

[Mue93] Frank Mueller. A library implementation of posix threads under unix. In *In Proceedings of the USENIX Conference*, 1993.

[NATC09] Angeles Navarro, Rafael Asenjo, Siham Tabik, and Calin Cascaval. Analytical modeling of pipeline parallelism. In *Proceedings of the PACT*, 2009.

[neh08] First the tick, now the tock: Next generation Intel microarchitecture (Nehalem). 2008. Intel White paper.

[ORSA05] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the International Symposium on Microarchitecture*, 2005.

[Phe08] Chuck Pheatt. Intel threading building blocks. *J. Comput. Sci. Coll.*, 23, April 2008.

[PLL$^+$06] J. Pisharath, Y. Liu, W.K. Liao, A. Choudhary, G. Memik, and J. Parhi. Nu-minebench 2.0. technical report. Technical Report CUCIS-2005-08-01, Center for Ultra-Scale Computing and Information Security, Northwestern University, August 2006.

[Poe05] Christian Poellabauer. Feedback-based dynamic voltage and frequency scaling for memory-bound real-time applications. In *In Proc. IEEE Real-Time and Embedded Technology and Applications Symposium*, 2005.

[QMS$^+$05] Carlos García Quiñones, Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2005.

[QSPK01] Daniel J. Quinlan, Markus Schordan, Bobby Philip, and Markus Kowarschik. The specification of source-to-source transformations for the compile-time optimization of parallel object-oriented scientific applications. In *14th Workshop on Languages and Compilers for Parallel Computing*, 2001.

[RCMA10] Brian W. Barrett Richard C. Murphy, Kyle B. Wheeler and James A. Ang. Cray user's group (cug), introducing the graph 500. May 2010.

[ROR$^+$08] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the Code generation and optimization*, 2008.

[RS01] Amir Roth and Gurindar S. Sohi. Speculative data-driven multi-threading. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, 2001.

[RVhRA08] Easwaran Raman, Neil Va hharajani, Ram Rangan, and David I. August. Spice: speculative parallel iteration chunk execution. In *Proceedings of the Code generation and optimization*, 2008.

[RVVA04] Ram Rangan, Neil Vachharajani, Manish Vachharajani, and David I. August. Decoupled software pipelining with the synchronization array. In *Proceedings of the PACT*, 2004.

[RWB09] Krishna K. Rangan, Gu-Yeon Wei, and David Brooks. Thread motion: fine-grained power management for multi-core systems. In *Proceedings of the 36th annual international symposium on Computer architecture*, 2009.

[SBV95] Gurindar S. Sohi, Scott E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the International Symposium on Computer Architecture*, June 1995.

[SLY$^+$11] Daniel Sanchez, David Lo, Richard M. Yoo, Jeremy Sugerman, and Christos Kozyrakis. Dynamic fine-grain scheduling of pipeline parallelism. In *Proceedings of the PACT*, 2011.

[Smi82] James E. Smith. Decoupled access/execute computer architectures. In *ISCA '82: Proceedings of the 9th annual symposium on Computer Architecture*, 1982.

[SMM$^+$09] Richard Strong, Jayaram Mudigonda, Jeffrey C. Mogul, Nathan Binkert, and Dean Tullsen. Fast switching of threads between cores. *SIGOPS Oper. Syst. Rev.*, April 2009.

[SOW$^+$95] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

[SPR00] Karthik Sundaramoorthy, Zach Purser, and Eric Rotenberg. Slipstream processors: improving both performance and fault tolerance. *SIGPLAN Not.*, 35(11), 2000.

[SQKP10] M. Aater Suleman, Moinuddin K. Qureshi, Khubaib, and Yale N. Patt. Feedback-directed pipeline parallelism. In *Proceedings of the PACT*, 2010.

[SQP08] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on cmps. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008.

[TCA07] William Thies, Vikram Chandrasekhar, and Saman Amarasinghe. A practical approach to exploiting coarse-grained pipeline parallelism in c programs. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, 2007.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *Proceedings of the 22nd annual international symposium on Computer architecture*, 1995.

[Tul96] Dean M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Proceedings of the 22nd Annual Computer Measurement Group Conference*, December 1996.

[VRR$^+$07] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative decoupled software pipelining. In *Proceedings of the PACT*, 2007.

[VSG$^+$10] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, 2010.

[WWDS94] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, 1994.

[XMM03] Fen Xie, Margaret Martonosi, and Sharad Malik. Compile-time dynamic voltage scaling settings: opportunities and limits. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, 2003.

[ZCT05] Weifeng Zhang, Brad Calder, and Dean M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, September 2005.

[ZCT06] W. Zhang, B. Calder, and D.M. Tullsen. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *International Symposium on Code Generation and Optimization*, March 2006.

[ZS01] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *Proceedings of the International Symposium on Computer Architecture*, July 2001.

[ZTC07] W. Zhang, D.M. Tullsen, and B. Calder. Accelerating and adapting precomputation threads for efficient prefetching. In *Proceedings of the International Symposium on High Performance Computer Architecture*, January 2007.