# Addressing Processor Over-provisioning on Large-scale Multi-core Platforms

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

in partial fulfillment

of the requirements for the degree

Doctor of Philosophy

by

Wei Wang

August

2015

APPROVAL SHEET

The dissertation

in submitted in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Wei Wang

---

AUTHOR

The dissertation has been read and approved by the examining committee:

Mary Lou Soffa

---

Advisor

Jack Davidson

---

Advisor

Kevin Skadron

---

Bruce Childers

---

Benton Calhoun

---

Accepted for the School of Engineering and Applied Science:

Craig H. Benson, Dean, School of Engineering and Applied Science

August
2015

**Abstract**

Modern micro-architectures have embraced multi-core processors and thread-level parallelism for performance growth, because of the difficulty of increasing single core performance without significantly increasing processor power consumption. To meet the ever growing need for speed, current large-scale computing platforms are Non-uniform Memory Accesses (NUMA) architectures equipped with dozens of cores, while the prediction is that future large scale systems will have hundreds or even thousands of cores. The applications executing on these platforms are usually multi-threaded applications which create large numbers of threads to simultaneously utilize the massive numbers of cores.

When executing multi-threaded applications on large-scale platforms, users and run-time systems typically allocate all available cores to their applications. However, because of the insufficient memory bandwidth on these large-scale platforms, many multi-threaded applications achieve their best performance when using only a portion of all available cores. Allocating all cores over-provisions these applications, degrades performance, and reduces energy efficiency and system throughput. Therefore, it is desirable to execute multi-threaded applications with the minimal core allocation that achieves best performance. We call this allocation, the optimal core allocation. However, determining the optimal core allocations for various applications on various hardware is very difficult.

Because memory bandwidth is the primary determining factor of optimal core allocations, we chose to predict optimal core allocations based on the prediction of memory bandwidth usage of multi-threaded applications. Accurately predicting memory bandwidth usage faces three major challenges: the random contention and concurrency in DRAM; the inter-processor connections with unknown properties; and the heterogeneity within the memory system. We thoroughly analyzed the memory system on large-scale NUMA platforms and discovered three important insights. First, DRAM contention and concurrency have stable statistical distributions. Second, inter-processor connections act like networks and have linear properties. Third, the memory system can be modeled as an integer problem. These insights allowed us to utilize probability theory and Mixed Integer Programming to predict memory

bandwidth usage with high accuracy and low-overhead. Based on the bandwidth prediction, we are able to highly accurately predict optimal core allocations in a short amount of time.

Our models predict optimal core allocations during application execution. Therefore, applying our models requires an efficient technique to dynamically adapt an application to its optimal core allocation during run-time. Moreover, our models require run-time information about application memory behavior and hardware configuration to make predictions. We designed a run-time technique that allows multi-threaded applications to efficiently adapt to their optimal core allocations during execution. To ensure low overhead and achieve near-ideal load-balancing on any core allocation, this run-time technique employs massive concurrent threads and distributed synchronization primitives. We also designed a low-overhead framework to support memory behavior profiling and hardware configuration detection at run-time by querying hardware performance counters and registers.

Combining the models and run-time techniques, we developed the OptiCore run-time system to automatically execute multi-threaded applications with their optimal core allocations on large-scale NUMA platforms. Compared to use-all-cores allocations, OptiCore provides a maximum speedup of 4.84. Additionally, the minimal core allocation used by OptiCore only allocates 12.5% of all cores. On average, OptiCore allocates only 88.7% of all cores and achieves 34.6% performance improvement.

# Acknowledgements

The seven years that I spent in graduate school were the most difficult time I have ever experienced. There were times that I almost quit. However, I was lucky that I have so many people supporting me and helping me. Without them, this dissertation would never exist.

I would like to express my deepest gratitude to my advisers, Mary Lou Soffa and Jack Davidson. Without their guidance and help, I would never be able to finish my dissertation. Their support goes far beyond research. They are my advisers, they are my best friends and they are my second parents. They established a role-model that I would like to follow for rest of my life. I am extremely lucky to have them as my advisers.

I would like to thank Kevin Skadron for his advice on research and career development. Most of my knowledge on modern computer architecture was from his teaching. I also want to thank him for mentoring wonderful students who also helped me a lot with my research.

I would like to thank Bruce Childers for his guidance on my research. We collaborated on our first project, which led me into the world of research and problem solving. I learned a great number of research skills from him. I hope we will be able to collaborate in the future.

Tanima Dey was my closest research colleague and friend in graduate school. We shared our research lab, we shared research ideas, and we shared defeat and success. I thank her for always giving me insightful feedback, and for the countless hours she spent on reading my bad English. I also would like to thank Jason Mars and Lingjia Tang for their help on research during my most difficult years in graduate school. Without their close assistance, I would never know how to write a good paper. I thank Jason Hiser for constantly answering my questions about compiler and assembly language. He is one of most tech-savvy people I have ever worked with.

I would like to thank Ryan Moore, Mary Jane Irwin, Mahmut Kandemir and Mahmut Aktasoglu for their help on my first project. It was such a pleasure to work with these brilliant people. I also want to thank Benton Calhoun for his rigorous review of my dissertation and my research, and for his valuable feedback.

The Computer Science Department at University of Virginia has many great graduate students. I particularly want to thank In Kee Kim, Runjie Zhang, Liang Wang, Nathan Brunelle, Weikeng Qin, Che Shuai and Jiayuan Meng for their help on my

research.

The companionship from my friends was the major reason that I survived. I any truly grateful to my friends, including but not limited to, Chih-hao Shen, Jian Xiang, Jing Yang, Kristen Walcott-Justice, Samyukta Jadhwani, John Hott, Puqing Wu, Ren Xu, Zhiheng Xie, Ming Mao, Vidyabhushan Mohan, Ke Wang, and Jin Qian. I start to realize that it is impossible for me to put every one of my friends here. Thank you all, my friends.

I acknowledge that this work is supported by National Science Foundation grants CCF-0811689 and CNS-0964627, and Air Force Research Laboratory (AFRL) contracts FA8750-13-2-0096 and FA8750-15-2-0054.

In the end, I want to thank my family for their understanding. Attending graduate school has such a huge impact to their life that I had never been able to foresee. I am still wondering, at nights, would their life be better if I had never gone back to school. I know they were as anxious as I was during the past seven years. Thank you for constantly being patient and supportive.

*To every graduate student*
*who is struggling, or has struggled,*
*with their research and life.*

# Contents

[1]This project was conducted jointly with Dr. Bruce Childers and Dr. Ryan Moore from University of Pittsburgh, as well as Dr. Mary Jane Irwin, Dr. Mahmut Kandemir and Mr. Mahmut Aktasoglu from Pennsylvania State University.

# Chapter 1

# Introduction

For the past three decades, the growth of computing performance has followed two powerful observations: Moore's Law and Dennard Scaling. Moore's law states that for every 18 months, the size of the transistors shrinks and the number of transistors on a new generation chip doubles [101]. Dennard Scaling states that, despite the doubling of transistors on a new chip, its power consumption is reduced because of smaller transistors [37]. This reduction in power consumption permits design of new processors with increasing clock frequency. That is, thanks to Moore's Law and Dennard Scaling, every new generation of processors has more transistors and higher frequency, allowing commensurate performance improvement, without increasing power consumption, and with no software changes required.

However, while Moore's Law continues to hold to some extent, Dennard Scaling has experienced gradual failure [22]. Because of the difficulty of continuously increasing the frequency without significantly increasing power consumption, processor designers have shifted to increasing core count (multi-core processors) to continue exploiting Moore's Law [49]. With these multi-core processors, the focus of performance improvement has switched from single core execution to the parallel execution on multiple cores. In the wake of this shift, software applications are switching to the multi-threaded execution paradigm – executing multiple threads on multiple cores to increase application execution speed. During the past several years, more and more applications, from high performance computing applications and data center applications, to general-purpose applications, and to even computer games, have switched to a multi-threaded execution approach [10, 15, 52, 69, 75, 135, 150, 153].

Driven by the ever growing need for speed, architectures and applications are now aiming at many-core and massive parallelization for performance improvement [23].

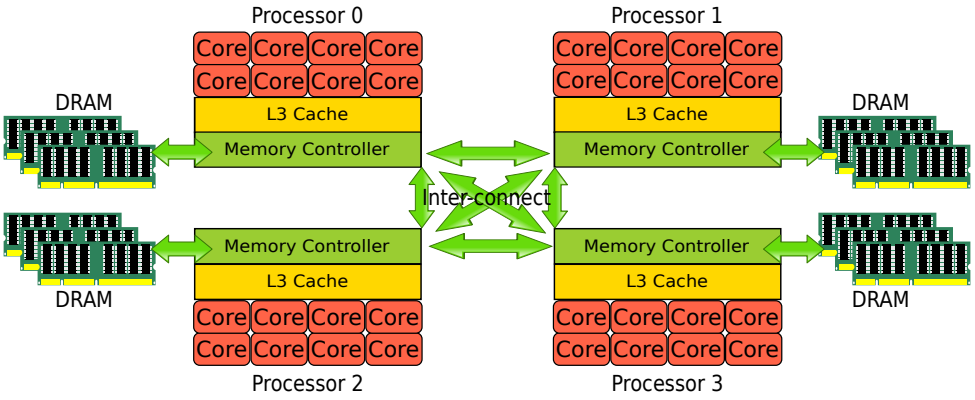


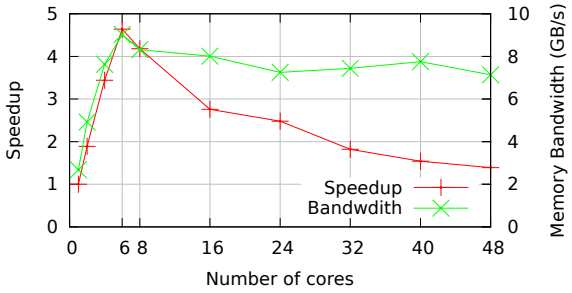

Figure 1.1: A NUMA machine with four-processors/eight-nodes.

That is, dozens, hundreds or even thousands of cores will be available, and a comparable number of threads will be executed, in one machine. However, the increased core/thread count renders other hardware resources scarce, and in particular, memory bandwidth is a limited resource [23]. Because DRAM speed is significantly slower than processor speed, one DRAM connection is not large enough to satisfy the bandwidth need of machines with large numbers of cores and threads.

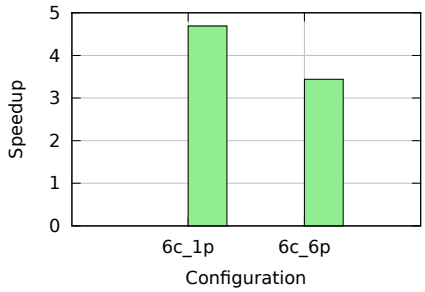
To address this discrepancy, non-uniform memory access (NUMA) architectures have been developed [61]. A modern large-scale NUMA machine typically consists of multiple multi-core processors, with each processor having its own DRAM connection. Consequently, multiple DRAM modules can be accessed simultaneously, allowing a much higher aggregated bandwidth. Figure 1.1 is a sketch of a NUMA machine with four processors. In this figure, because each of the four processors has its own DRAM connection, the aggregated memory bandwidth is four times larger than a single DRAM connection shared by all four processors. Additionally, as illustrated in Figure 1.1, processors on NUMA machines are also connected together using inter-processor connections so that a thread running on one processor can access the memory of another processor. These inter-processor connections provide the illusion of a single DRAM connection shared by all processors, which greatly simplifies the programming of multi-threaded applications.

## 1.1 The Processor Over-provisioning Problem

When executing a multi-threaded application on a NUMA machine, a key issue is to determine the number of cores that should be allocated to an application. Currently,




(a) Speedup (over using one core) and memory bandwidth usage of *streamculster* using 1 to 48 cores.




(b) Speedup comparison: 6 cores on 1 processor (6n_1p) V.S. 6 cores on 4 processors (6c_4p)

Figure 1.2: Speedup (over using one core) of PARSEC benchmark *streamcluster* running on the NUMA machine in Figure 1.1.

users and run-time systems tend to allocate all available cores for the execution of their applications, based on the naive assumption that more cores translates to increased performance [7, 68, 69]. Unfortunately, despite the huge aggregated memory bandwidth increase on NUMA machines, the bandwidth is still limited and is still insufficient for some applications. This insufficiency can significantly limit the scalability of multi-threaded applications and cause performance degradation when using large numbers of cores and threads.

Figure 1.2a shows the performance of a PARSEC benchmark *streamcluster* running on a 48-core NUMA machine, using one to forty-eight cores [15]. This NUMA machine has four processors, with each processor containing twelve cores. As the figure shows, *streamcluster* has the highest speedup of 4.64x when using six cores, while the speedup of using all forty-eight cores is only 1.39x. In other words, *streamcluster* only requires six cores to achieve its best performance on this NUMA machine. Allocating more cores than necessary, or *over-provisioning processors*, degrades performance.

The primary scalability limitation that prevents *streamcluster* from achieving higher performance when using more than six cores is memory bandwidth. Figure 1.2a also shows the memory bandwidth usage of *streamcluster*, which peaks when using six cores. After the maximum bandwidth, or the bandwidth limit, is reached, adding more cores does not improve performance because the hardware cannot provide data fast enough to satisfy the needs of the additional cores. In fact, using additional cores degrades performance, as the extra memory resource contention introduced by these additional cores reduces the total bandwidth usage. Furthermore, the additional threads increase the amount of communication among the threads and

degrade performance.

More importantly, core allocation on large-scale NUMA platforms is not only about the total number of cores, but also about the location (which processor) of the cores. Figure 1.2b shows the speedup of *streamcluster* using six cores on the same NUMA machine with two different allocation configurations. In the first configuration, all six cores are allocated on one processor. In the second configuration, the six cores are allocated on four processors. As the figure shows, although the two configurations have the same core count, their performance differs considerably. This performance difference is caused by *streamcluster*'s communication pattern. *Streamcluster* uses more memory bandwidth on some processors than the others. Therefore, the cores on processors with high bandwidth usages need to be allocated differently from those on processors with low bandwidth usages.

In addition to negatively impacting performance, processor over-provisioning negatively impacts power efficiency and system throughput. Although the pipelines of the over-provisioned cores are stalled for data, the cores are still in a high power state and consume considerable amount of energy. Therefore, processor over-provisioning also leads to poor energy efficiency [39, 88]. Furthermore, over-provisioned cores can be released to execute other applications to boost overall system throughput and improve system utilization.

Because processor over-provisioning degrades performance, energy-efficiency, throughput and system utilization, it is important that multi-threaded applications are executed with the minimum core allocations in the right configuration that maximizes their memory bandwidth usages and performance. We call such a core allocation the *optimal core allocation*.

The overall goal of this research is to provide a practical solution that can automatically execute large-scale multi-threaded applications with their optimal core allocations on any large-scale NUMA platform.

Note that, besides memory bandwidth, there are other factors that affect the scalability of multi-threaded applications. Notably, synchronization, cache contention and insufficient parallelism can affect scalability. Nonetheless, our experimental results show that, for large-scale applications that are typically executed on large-scale NUMA platforms, memory bandwidth is the primary limitation. There are many studies addressing the problem of synchronization, cache contention and insufficient parallelism [25, 54, 56, 60, 79, 86, 99, 123, 132, 144]. However, the scalability limitation of memory bandwidth on large-scale NUMA machines has not been adequately

addressed. Consequently, this research focuses on the memory bandwidth limitation. The studies on the other scalability limitations can be used in conjunction with this research to provide a comprehensive solution for multi-threaded applications of any scale.

## 1.2 Research Challenges

To achieve the goal of automatically executing multi-threaded applications with their optimal core allocations on large-scale NUMA machines, there are five challenges to overcome. These challenges involve determining the memory bandwidth usages, the optimal core allocations, as well as the efficient execution and adaptation of multi-threaded applications with a run-time system.

### 1.2.1 Challenge 1: Prediction of Memory Bandwidth Usage

For large-scale applications on NUMA platforms, scalability is primarily limited by memory bandwidth. Consequently, determining the optimal core allocation involves predicting the memory bandwidth of an application. Indeed, the memory bandwidth usage of a multi-threaded application needs to be predicted before the prediction of its optimal core allocation. For accurate optimal core allocation prediction, the memory bandwidth usage must be accurate.

Unfortunately, accurately predicting memory bandwidth usage is very difficult because of the complex interference and contention among concurrently executing threads. On a NUMA platform, threads share various memory resources, including the DRAM modules, the DRAM connections, and the inter-processor connections. These threads contend for these shared memory resources, which significantly affects the actual memory bandwidth usage obtainable. Consequently, to accurately predict the memory bandwidth usage, the impact of the memory resource contention must be precisely determined. Additionally, the severity of the memory resource contention varies considerably with the application, input set, and hardware configuration.

There has been several approaches for predicting the memory bandwidth usage of a multi-threaded program [30, 60, 81, 86, 144, 164]. However, these approaches are inaccurate, because they did not consider contention in the memory resources.

### 1.2.2 Challenge 2: Prediction of Optimal Core Allocation

With the accurate prediction of memory bandwidth usage, optimal core allocation can be determined naively by comparing the bandwidth usage of different core allocations and selecting the minimum core allocation with the highest memory bandwidth. However, this comparison-based approach can be heavy-handed. As stated in section 1.1, even though two core allocations have the same number of cores, the two allocations can still have significantly different performance and bandwidth usages, if the locations (processors) of the cores are different. As a result, on a large-scale NUMA machine, there can be millions of core allocations to compare. For example, on an existing AMD NUMA machine of eight processors with six cores each, the total number of possible core allocations is $7^8 = 5,764,801$. Comparing millions of core allocations is impractical. Therefore, a technique must be designed to determine the optimal core allocation from millions of candidates within reasonable amount of time.

To the best of our knowledge, there is no research that predicts the optimal core allocation by predicting memory bandwidth usage. However, there have been studies that directly predicted optimal core allocations without predicting the memory bandwidth [31, 39, 77, 114, 123, 138]. These studies failed to accurately predict optimal core allocations because they did not considered the memory resource contention. Some of these studies also have high overhead because of the heavy-handed search for the optimal solution.

### 1.2.3 Challenge 3: Run-time Core Reallocation

Because optimal core allocations vary with the application, input set and hardware configuration, they are best (most accurately) predicted using the actual application memory behavior obtained during execution. This observation implies that a run-time solution is most suitable to solve the processor over-provisioning problem. With a run-time solution, a multi-threaded application will start with a suboptimal core allocation for memory behavior profiling. After the profiling and the optimal core allocation prediction, the application can then be adapted to use this optimal allocation. This adaption of core allocation during execution requires a solution that can remap the executing threads to any core allocation at run-time. Additionally, many multi-threaded applications have phases during their execution. These phases usually have different memory behavior, and thus require different core allocations. The changing of core allocation from one phase to another also requires an efficient

run-time reallocation solution. Unfortunately, the rigidity of current thread libraries prevents multi-threaded applications from being reconfigured to use the optimal core allocation at run-time without considerable performance penalty.

Consider the case where a multi-threaded application runs on a machine with 16 cores. Initially, this application uses all 16 cores with 16 threads. After a period of time, it is predicted that 10-core allocation is the optimal. Therefore, the application has to adapt to 10 cores. Ideally, the application should be reconfigured to run with 10 threads. However, running 16 threads on 10 cores can cause an unbalanced load on the cores and performance degradation. Furthermore, many user programs are rigid in that their thread count cannot be changed at run time. The inability to efficiently reconfigure a multi-threaded application limits previous run-time core allocation solutions to only data-parallel loops, where the workload can be easily repartitioned after one loop iteration ends [32, 33, 60, 144].

There has been research on enabling multi-threaded applications to adapt to any core allocation during execution [86, 104, 138]. However, some of these techniques require modification of application source code, which can be difficult [86, 104]. Other techniques have high overhead on large-scale NUMA machines, because they introduce considerable synchronization overhead [86, 138].

### 1.2.4 Challenge 4: Low Overhead Run-time Solution

As stated in the previous challenge, addressing the processor over-provisioning problem is best solved with a run-time solution. This run-time solution needs the following three steps, at least:

1. application memory behavior profiling;
2. optimal core allocation prediction; and,
3. dynamic core reallocation.

A run-time solution inevitably adds overhead because it introduces an extra layer into the system to monitor and manage executing applications. These three steps can further increase this overhead. Therefore, it is important that these steps are carried out in a way that minimizes overhead as much as possible.

### 1.2.5 Challenge 5: No User-involvement and Source Code Independent

Many existing multi-threaded applications have complex code bases. If a solution of the processor over-provisioning problem requires source code modification, the difficulty of applying this solution to complex source code may prevent it from being employed. Furthermore, users of multi-threaded applications do not always have access to the source code of their applications. Consequently, in order for a solution to be applied to existing applications, this solution should be carefully designed so that it does not require modifications to, or knowledge of, application source code.

Moreover, as stated above, predicting the optimal core allocations requires detailed information about the underlying hardware configuration. However, the complexity and the variety of hardware configurations make it very difficult for ordinary users to comprehend these configurations. A solution that requires users to input the information of hardware configurations may prevent ordinary non-expert users from employing this solution. Hence, in order for the solution to be employed by ordinary users, this solution should be carefully designed so that it does not require user involvement.

## 1.3 General Approach

This dissertation presents a comprehensive solution to the processor over-provisioning problem on large-scale NUMA machines by addressing the challenges discussed in the previous section.

Because memory bandwidth is the determining factor of scalability and optimal core allocation on large-scale NUMA platforms, we chose to predict the memory bandwidth usage of an application before predicting its optimal core allocation, so that highly accurate optimal core allocation predictions can be achieved. That is, we develop two models: one predicts the memory bandwidth usage, while the other one predicts optimal core allocations based on the predicted bandwidth usage.

However, models cannot directly benefit ordinary users because they cannot be directly applied to applications with run-time support. Consequently, in this dissertation, we also provide run-time techniques to support the adoption of our models. Finally, we combine our models and run-time techniques into one run-time system which can automatically execute multi-threaded applications with their optimal core

allocations on large scale NUMA machines.

These models and run-time techniques are carefully designed and combined so that they have low run-time overhead, and they require no user-involvement and no source-code modification. In order to demonstrate that our solution is practical and readily-applicable, we evaluated our models, run-time techniques and the final run-time system on real large-scale NUMA platforms. More specifically, we developed the following models and run-time techniques to address the challenges discussed in Section 1.2:

1. A model that predicts the memory bandwidth usages of multi-threaded applications when they execute with different core allocations. Because bandwidth usage is determined by the memory resource contention, this model first predicts the contention at DRAM module, DRAM connection and inter-processor connection. Then, based on the contention prediction, the model predicts the memory bandwidth usages. Because memory resource contention is explicitly modeled, this two-step procedure ensures highly accurate prediction results.

2. A model that predicts the optimal core allocation of multi-threaded applications. This model makes predictions using the results from the above bandwidth usage model. In order to find the optimal solution from millions of possible core allocations, Mixed Integer Programming (MIP) is employed. To be more specific, this model converts the optimization goal (maximize bandwidth and minimize core allocation) and the bandwidth usages into linear functions. Then it uses an MIP solver to find the optimal core allocation.

3. A run-time technique that remaps the application threads to a new core allocation efficiently during execution. This technique makes use of the fact that current multi-threaded applications can partition their work into fine-grained small jobs when they are instructed to create and utilize a large number of threads, with one thread assigned to process one job. When a multi-threaded application adapts to a new core allocation, this technique redistributes all jobs to this new allocation in a way that ensures each core gets similar load (similar number of jobs). Because one thread is responsible for one job, redistributing a job to a new core is equivalent to re-mapping its corresponding thread to the new core. The downside of creating numerous threads is that it increases the number of synchronization operations. This increase can introduce significant

overhead. To reduce this overhead, distributed synchronization primitives are employed.

4. A run-time framework that communicates with the operating system (OS) to provide basic services that are required by the dynamic core reallocation technique. This framework also provides services to profile application memory behavior. These profiling data are used as inputs by the bandwidth and optimal core allocation models. This framework is also responsible for detecting the phase changes within an application. A phase change may alter the application's memory behavior, and potentially calls for a different core allocation. This run-time framework is carefully designed such that it has low overhead and can be applied to existing application without source code modification.

## 1.4 Contributions

The contributions of this research include the models for the prediction of memory bandwidth usage and optimal core allocation, the novel run-time techniques and framework for on-line performance profiling and application execution management, the complete run-time solution for the processor over-provisioning problem, as well as the experimental evaluations of our models and solutions on two real large-scale NUMA platforms with 19 benchmarks. The detailed contributions are described as follows.

### 1.4.1 Contribution 1: The Memory Bandwidth Model

The first contribution is a practical and highly accurate memory bandwidth model, DraMon, for predicting the DRAM module contention, DRAM connection contention and memory bandwidth usages of multi-threaded applications. DraMon was evaluated on a large-scale NUMA machine with AMD processors, using 22 benchmarks from PARSEC and NBP benchmark suites [15, 75]. Experimental results show that DraMon has an average accuracy of 98.6% for DRAM contention predictions, and an average accuracy of 93.4% for memory bandwidth usage predictions. Additionally, DraMon has very low overhead, and requires a short amount of time to make predictions.

In addition to addressing the processor over-provisioning problem, DraMon can be used for other DRAM-related optimization research, such as providing perfor-

mance estimations for novel memory allocation algorithms and parallel algorithms. Moreover, the development of DraMon reveals that DRAM contention has a stable statistical distribution, allowing it to be quickly predicted. This dissertation also provides an analysis of the factors related to bandwidth usage and their application for bandwidth prediction.

### 1.4.2 Contribution 2: The Optimal Core Allocation Model

The second contribution is an optimal core allocation model, NuCore, for predicting the maximum bandwidth usages and optimal core allocations of multi-threaded applications on large-scale NUMA machines. NuCore was evaluated on two large-scale NUMA machines, using 19 PARSEC and NBP benchmarks [15, 75]. Experimental results show that NuCore has an average accuracy of 90% for memory bandwidth usage predictions. Our experiments also show that NuCore can correctly predict the optimal core allocation of 19 benchmarks out of a total of 22 benchmarks. For the other three benchmarks, NuCore's predictions differed by at most one core per processor from the experimentally determined optimal core allocation. Additionally, these results show that NuCore is very fast and has over low overhead.

These predictions can also be used by application developers, computer architects and system administrators to study and improve the performance of their large-scale applications and machines. The development of NuCore reveals interesting characteristics of the various types of memory contention on NUMA machines. NuCore is the first analytical model to mathematically express the NUMA bandwidth limiting factors and predict bandwidth usage and optimal core allocation. The technique of expressing non-linear memory bandwidth limiting factors and memory contentions as linear constraints, and expressing prediction or optimization goals as objective functions may also apply to other memory-related optimization problems.

### 1.4.3 Contribution 3: The Run-time Core Reallocation Technique

The third contribution is an efficient run-time technique, FlexThread, for dynamic core reallocation at run-time. FlexThread enables a run-time system to adapt a multi-threaded application to any core allocation efficiently. Experimental results from two large-scale NUMA machines with 19 PARSEC and NPB benchmarks show that FlexThread has a maximum overhead of only 5% [15, 75].

FlexThread's dynamic and efficient core re-allocation required by various run-time optimization algorithms. Additionally, FlexThread provides significant performance improvement even without optimization algorithms, because it helps applications improve cache and processor utilization, provides better load balancing, and employs distributed synchronization primitives. This better management provides up to 1.9 times speedup over GNU PThread and OpenMP libraries. FlexThread is carefully designed so that it is readily applicable to existing applications without the need to change application source code. This dissertation also provides detailed experimental and theoretical analysis of FlexThread's overhead to help users determine the best configuration of FlexThread for their applications and platforms.

### 1.4.4 Contribution 4: The Low Overhead Run-time Framework

The fourth contribution is a run-time framework, REEact, for writing user-specific, application-specific and platform-specific policies to manage applications and hardware resources. REEact provides a wide range of services to support run-time monitoring of application behavior and hardware status. It also provides services to adapt application execution, change resource allocation, and reconfigure hardware components. All these services are exported to users through easy-to-use Application Programming Interfaces (API). REEact is portable, and has low overhead. These services and properties make REEact ideal for testing, designing and deploying novel run-time systems. Experiment results on two large-scale NUMA machines with 13 PARSEC benchmarks show that REEact has a maximum overhead of only 3% [15].

### 1.4.5 Contribution 5: The Complete Run-time Solution

The last contribution is the complete run-time solution, OptiCore, which automatically executes multi-threaded application with their optimal core allocations. Combining DraMon, NuCore, REEact and FlexThread, OptiCore offers a complete, low-overhead, user-involvement-free and application-source-code-independent solution for the processor over-provisioning problem on large-scale NUMA platforms.

When evaluated on two large-scale NUMA platforms with 19 benchmarks from PARSEC and NAS Parallel benchmark (NPB) suites, OptiCore provides an average performance improvement of 34.6%, over use-all-cores allocations. The maximum speedup is 4.84 times. Moreover, the overhead of OptiCore is less than 5%.

## 1.5 Dissertation Organization

The rest of this dissertation is organized as following: Chapter 2 discusses related work. Chapter 3 discusses NUMA architectures and the impact of memory bandwidth on scalability. Chapter 4 presents the DraMon model which predicts DRAM contention, DRAM request overlapping and local memory bandwidth usage. Chapter 5 presents the NuCore model which predicts total memory bandwidth usage and optimal core allocation. Chapter 6 presents the FlexThread technique for efficient run-time core reallocation. Chapter 7 presents the REEact framework for run-time application monitoring and management. Chapter 8 presents the OptiCore run-time system that automatically executes applications with their optimal core allocation to address the processor over-provisioning problem. Finally, Chapter 9 summarizes this research and provides directions for future exploration.

# Chapter 2

# Related Work

This chapter discusses related research work. Research on addressing the processor provisioning problems is discussed first, after which, research related to each component of OptiCore is discussed.

## 2.1 Processor Over-provisioning

**Scalability Analysis**

These are many studies on analyzing the scalability of multi-threaded applications. These studies reveal that many multi-threaded applications suffer performance loss when using large numbers of cores and threads. Bienia et al. investigated the scalability of PARSEC benchmark suites [16]. However, because of the small scale machine used, they found PARSEC benchmarks scaling well. Heirman et al. analyzed the scalability of several multi-threaded programs using cycle stacks [59]. They found memory bandwidth, synchronizations and unbalanced workloads are the major scalability-limiting factors. Mandal et al. analyzed scientific workloads and concluded that these workloads could easily saturate the memory bandwidth on multi-core platforms [96]. Pusukuri et al. studied the impact of synchronizations and context switches [123]. They discovered that synchronization operations could significantly limit scalability when large numbers of threads were used. Du Bios et al. proposed a novel tool set to visualize scalability bottlenecks [42]. They analyzed Java applications using the new visualization tool set, and discovered that data sharing among garbage collector threads might limit scalability. Liu et al. developed a new tool for analyzing the performance bottleneck of multi-threaded applications on NUMA platforms [90]. They

reported that memory bandwidth is the primary scalability limitation on NUMA platforms. A similar observation was also made by Boyd-Wickizer et al. on large-scale many-core systems [23]. These studies on scalability corroborate our finding that memory bandwidth is the primary scalability limitation. These studies provide valuable insights which helped us design our models.

### Predicting Optimal Core Allocations

There are also several studies focused on determining optimal core allocations for multi-threaded applications. One group of studies employed a search-based solution. Nyuyen et al. proposed a framework to dynamically detect the optimal core allocation by searching and sampling different core allocations [114]. To reduce the number of allocations to sample, they used an extrema searching algorithm called the Method of Golden Sections. Corbalan et al. proposed a similar searching solution that tried different core allocations [31]. To reduce the number of trials, they set a performance target, and would stop searching when the target was met. Pusukuri et al. proposed a method that searches different core allocations and thread counts for the optimal one based on OS observations, such as context switch counts and synchronization operations [123]. Unlike other research, Pusukuri el al. focused more on the search of optimal thread counts instead of the optimal core count. Similar to our research, they also observed that executing more threads than cores could be beneficial for certain applications. Li et al. also used a search-based algorithm to dynamically change core allocation to reduce power consumption [88]. They employed binary search and hill-climbing optimization to prune the search space. Depending on the strategies employed to reduce sample counts, the search-based techniques either could not predict the optimal core allocation (up to 24% slower than the optimal), or incurred high searching overhead (up to fifty samples required). Because of the high overhead and the mispredictions, our research in this dissertation does not use the search-based approach.

Another group of studies employed a regression-based solution. Jung et al. built a model based on regression to predict the performance for a particular core allocation [77]. This model used instructions per cycle (IPC) and cache misses as inputs. Ding et al. also employed regression to predict the performance and energy-delay-product (EDP) for different numbers of cores [39]. This work primarily focused on energy consumption instead of performance, and it did not aim at optimal results. Sridharan et al. proposed a resource-agnostic model based on Amdahl's law to predict

optimal thread count [138]. This work only predicted the total thread count and did not consider the non-uniform memory topology of NUMA machines. Sasaki et al. investigated node allocations on NUMA machines [130]. They sampled the instructions per second for different core allocations. Then they built a model based on Amdahl's law to predict to optimal numbers of nodes. In conclusion, the authors predicted that allocations based on cores instead of nodes might provide better performance, which corroborates our findings. Limited by the number of available samples and the hardware resources considered, these regression-based solutions usually cannot provide an accurate prediction. The predictions they made were up to 65% slower than the optimal core allocations. To provide more accurate predictions, our research individually considers the hardware resources which potentially limit scalability, rather than combining them into one regression model.

The last group of studies developed a model for each scalability-limiting factor. Suleman et al. investigated the scalability-limiting factors of memory bandwidth and critical sections [144]. They provided a predictive model for each factor. For memory bandwidth, a simple linear model was employed. Lee et al. studied the scalability-limiting factors of memory bandwidth and cache size [86]. However, they also used linear models for memory bandwidth. Heirman et al. investigated under-subscription of threads on SIMD processors due to cache contention [60]. Because SIMD architecture is drastically different than NUMA architecture, their approach cannot be applied to NUMA platforms. More importantly, the models developed by these studies are inaccurate because they failed to model the thread interference at DRAM and inter-processor connections, which are two primarily scalability limiting factors on large-scale NUMA machines. Because we carefully model the DRAM contention and inter-processor connection contention, our OptiCore run-time system has better performance.

Moore and Childers proposed a novel model to predict the thread counts of multiple co-running multi-threaded applications [102, 103, 107]. They first profiled each application offline to build a utility model. Then the models were used in multi-programmed workloads to determine the best thread count for each application for various optimization goals. There are two major differences between this work and our research. First, we choose a completely online (run-time) approach, while their work took an approach that combined both offline and online techniques. Second, we focus on optimal performance, while their work aimed at multiple optimization goals.

## 2.2 DRAM Contention and DRAM Bandwidth Modeling

OptiCore's component DraMon focuses on the modeling and prediction of DRAM contention and DRAM bandwidth. In the past years, there are several studies that modeled DRAM analytically. Choi et al. modeled the time that a DRAM bank was busy processing data [30]. Similar to our work, their model considered row buffer misses and hits. However, their model requires memory traces, and the goal was to provide guidelines for DRAM design rather than predicting bandwidth usage. Wang et al. analyzed the DRAM internal components and presented a simplified DRAM resource model for high-level performance analysis [152]. This resource model includes DRAM banks, row buffers, data bus and command bus. Although this resource model does not predict DRAM contention or bandwidth usage, it greatly helped us in the design of DraMon. Yuan et al. modeled the DRAM busy time on GPGPUs [164]. This model used application memory traces to predict the amount of time that a DRAM bank was servicing data. Kim et al. proposed a model to predict the impact of bank partitioning [80]. Ahn et al. modeled bandwidth usage of programs with regular memory access patterns [4]. These two models did not consider DRAM contention, which is the determining factor of DRAM bandwidth usage. Subramanian et al. proposed a performance model to estimate the slowdown due to DRAM contention, which required hardware modification [143]. Du Bois et al. proposed novel hardware counters to estimate performance interference from memory resource contention [41]. Adding these counters also requires modifying existing hardware. Additionally, all of these studies were conducted using simulators, while DraMon is evaluated on real machines and existing hardware.

A linear bandwidth model was first proposed by Snavely et al. for multi-processors systems [136]. This model assumed that the memory bandwidth usage increases linearly with the number of cores. It was later applied to multi-core systems [86, 127, 144]. To improve accuracy, the linear model was also extended with a roof-line, i.e., the maximum peak bandwidth [157]. Kim et al. proposed modeling bandwidth with multiple linear and logarithmic regressions [81]. This model using either linear or logarithmic regressions for different core allocations (more accurately, the numbers of cores). As demonstrated in this dissertation, both the linear model and regression-based model have low accuracy, because several important factors are overlooked.

Cache reuse distance has been used to predict cache miss rate by many studies.

The idea of whole program cache reuse distance was first used by Ding and Zhong to predict cache performance [38]. They predicted the cache performance of large input sets based on profiling data of smaller data sets. Chandra et al. were the first to employ cache reuse distance to address the cache contention problem on multi-core processors [26]. They sampled the cache reuse distances for different single-threaded programs. Then they used these samples to predict the cache contention among these programs. Xiang et al. extended this idea into online models, allowing more accurate run-time management of cache contention [159]. The authors of this work simplified the sampling of cache reuse distances, allowing run-time prediction of cache contention for multi-threaded programs. The contention in DRAM row buffer is similar to cache contention in that row buffers are also used to temporarily hold data and are also sensitive to data locality. These research efforts inspired us to use bank reuse distance in DraMon. Note that, the different between DRAM and cache allows us to predict DRAM contention with extremely low overhead, which is currently impossible for cache contention prediction.

## 2.3 NUMA Memory Bandwidth Modeling

Besides predicting optimal core allocation, OptiCore's component NuCore also predicts the inter-processor and total memory bandwidth usages of multi-threaded applications on large-scale NUMA machines. To the best of our knowledge, there is no prior research that models the inter-processor memory bandwidth. However, there are several studies that analyzed the impact of inter-processor memory accesses. In a recent work, Blagodurov et al. discovered that the contention for the inter-processor memory connection could impact the memory bandwidth usage on that connection [19]. Because of their work, we paid special attention to the contention at the inter-processor memory connections. Majo et al. further noticed that the outgoing memory bandwidth from a processor is affected by the processor's local memory accesses [92–95]. Because of these studies, we considered the contention among local and remote memory accesses in NuCore.

Additionally, NuCore employs Mixed Integer Programming (MIP) to make predictions. MIP has long been used to solve scheduling problems on computer systems with linear properties [84, 151]. Nowatzki et al. proposed a generic MIP framework for scheduling programs on spatial architectures [117]. These studies inspired us to employ integer programming. However, Nowatzki et al. pointed out that MIP-

based solutions were limited to problems with "directly expressible" and linear-only constraints [117]. However, the core of the memory bandwidth problem — the dynamic contention in the memory system — is neither linear nor directly expressible. Our insights of NUMA characteristics in Chapter 5 and our technique of converting non-linear constraints to linear make it possible to apply MIP to memory-related problems.

## 2.4 Dynamic Core Reallocation

Previous research that requires dynamically reconfiguring a program to its optimal core allocation is limited to data parallel loops, where workload repartition is permitted at the boundary of loop iterations [32, 33, 39, 77, 114, 137, 144]. Moore and Childers proposed a method to dynamically reconfigure an application to use any number of cores at run-time [104]. They proposed a novel inflate/deflate programming model which allowed run-time system to dynamically control the parallelism of an application. The proposed programming model enables various application-specific management techniques during execution. However, this method requires modification to a program's source code. Lee et al. developed a solution to dynamically reconfigure a program without source code modification [86]. Similar to our research, they proposed to create large numbers of threads to help load-balancing various core allocations. Nonetheless, this solution still needs source code to analyze communication patterns. It also requires expensive off-line profiling. However, in order to work with existing applications, we believe it is better to directly manage binary executables, because it may not always be feasible to modify existing source code. Sridharan et al. proposed a dynamic core reallocation solution that does not require source code modification [138]. However, their solution increases the number of synchronizations performed by an application, and thus degrades performance. FlexThread in this dissertation, on the other hand, provides a low-overhead dynamic core reconfiguration solution that does not require application source code.

## 2.5 Run-time System and Virtual Execution Environments

Similar to REEact framework, there has been much work in designing and implementing run-time systems and virtual execution environments (VEEs) for various purposes. Nesbit et al. proposed virtual private machine (VPM) as an abstraction for managing spatial and temporal resources in multicore systems [113]. These VPMs consisted of several software policies for managing resources which translated system-level requirements into different hardware mechanisms. Cuvillo describes a Thread Virtual Machine (TVM) in the form of a thread library to allow applications to achieve full resource utilization [35]. TVM aimed at providing user-level hardware resource management for Cyclops-64 many-core processors. AKULA was a tool-set for experimenting and testing cache-contention-aware scheduling algorithms [169]. AKULA was primarily aimed at providing performance estimations for scheduling algorithms on future large-scale many-core processors. Noll et al. described a virtual machine called CellVM to allow programmers to use higher-level programming constructs and mimic the behavior of a homogeneous shared memory multiprocessor, hiding the heterogeneity in the underlying Cell processors [116]. Because efficiently programming Cell processors required detailed understanding of this architecture, OpenMP was used to decouple the low-level architecture resource management from application algorithms. CellVM aimed at providing a novel middle-ware to replace OpenMP with better abstraction and performance. Similar to these researchers, we advocate the use of run-time systems and VEEs to dynamically manage applications and hardware resources on multi-core and many-core platforms. However, these VEEs are usually specialized for a particular purpose (mostly for shared resource management) or an architecture, while REEact is designed to provide a generic framework and a wide variety of services that can be used for a range of diverse purposes.

The Multikernel Operating Systems shared the same application and architecture-specific management spirit with us [13]. However, that work focused on improving the OS kernel design instead of providing a flexible run-time on top of existing OS, which is the approach taken by REEact. Log-based architecture (LBA) is similar to REEact in that LBA also constantly monitors on-line applications [28]. The difference is that LBA focuses more on application security and correctness. REEact shares the concept of user-level application management with scheduler activation [8] and shares the spirit of application-specific resources management with exokernel [47]. However,

for multi-core platform resource/application management, extensive user-level/kernel-level thread interactions and application-specific resource abstractions are not always necessary.

There has been prior work on virtualization and hypervisors. Xen is a hypervisor that allows the existance of large number of guest OSes on the same machine with low overhead and safe resource isolation [12]. In another work, the Xen hypervisor was extended to create a framework called VT-ASOS, for application-specific resource management in many core systems [115]. VT-ASOS allowed the incorporation of application-specific components, such as schedulers and memory allocators, into traditionally hypervisors. Haase et al. described a scalable dataflow-driven and self-distributing virtual machine for multicore-FPGAs [57]. This virtual machine allowed transparent runtime configuration of the underlying hardware to adapt dynamically to the changing environment. Kauer et al. presented a virtualization architecture consisting of a microhypervisor and environment that provides operating system functionality including virtual-machine monitors (VMM) in user-level [140]. This user-level VMM enabled the execution of unmodified guest OSes in the virtual machine. Unlike these studies, REEact focus on run-time management of applications (instead of OSes) and provides supports for customizable user-level resource management policies.

## 2.6 Other Related Work

There are studies investigating the application affinity problem, which is related to the core allocation problem. Moore and Childers et al. proposed a novel online technique called AutoFinity to dynamically determine the application affinity (thread-to-core mapping) for multi-threaded applications [105]. AutoFinity utilizes machine learning methods to construct an action table which provides hints for thread-to-core mappings. During execution, applications are first profiled using PMUs. Then the profiling data are used to locate the corresponding hints for the application, and a proper thread-to-core mapping is determined based on the hints. In a recent work, Moore and Childers also proposed to build application-specific and thread-count-specific models for the application affinity problem [106]. These offline models are more accurate because they consider a wide ranges of features (properties of the hardware and software), and they consider the varying requirements of different applications and thread counts. The problem of application affinity is different than the problem of

core allocation. The optimal core allocation problem requires determining how many cores (threads) to use, while the application affinity problem requires determining which cores to use given a specific number of threads. Additionally, our model can be modified (by adding another constraint about total core count) to address the application affinity problem when memory bandwidth is the primary factor that impacts the mapping of threads.

There are several studies aimed at improving memory allocation algorithms to mitigate DRAM contention. Liu et al. improved Linux memory allocation algorithm [89]. They employed page-coloring to partition DRAM banks and their associated memory pages among co-running threads. Mi et al. studied the potential (i.e., maximum speedup) of bank partitioning [100]. They observed that with ideal hardware and software, up to 14% speedup could be achieved. Park et al. observed that memory requests with regular access patterns degraded performance because of DRAM contention [118]. Based on this observation, they argued that memory allocation algorithms should allocate a new page from a randomly selected bank. Because of the energy limitation, only four banks in a DRAM module can be active simultaneously. Therefore, these techniques cannot dramatically reduce DRAM contention and increase total memory bandwidth usage. Hence, they cannot eliminate the bandwidth saturation problem on large-scale NUMA platforms. Additionally, our memory models can be revised to predict the DRAM contention and bandwidth usage under these new allocation algorithms by updating their functions that consider memory allocation schemes.

Observing that inter-processor memory accesses affect performance, many studies aimed at reducing the number of these accesses. Awasthi et al. improved data placement with adaptive first-touch policy and page migration on NUMA machines [11]. This work dynamically analyzed page accesses to allocate a page only on the node that was predicted to use it. And it migrated pages if remote accesses were frequent. Dashti et al. proposed using memory page migration and replication to address memory traffic congestion [34]. They migrated a page to the node that actually accessed it, or duplicated a page if multiple nodes used it. These two techniques aim to reduce memory latency rather than reducing bandwidth usage. In fact, the reduced memory latency improves performance and potentially increases bandwidth usage. Therefore, OptiCore is complementary to these techniques.

Novel hardware were also proposed to improve memory system. Fang et al. proposed a new memory controller design to reduce the number of coherence messages

over inter-processor connections [51]. This new design could increase the maximum available bandwidth of inter-processor connections. Hardavellas et al. designed techniques to optimize the data placement on Non-Uniform Cache Access (NUCA) architectures [58]. They categorized applications based on their data access patterns, and dynamically chose data placement strategies for different patterns. Ipek et al. proposed using reinforcement learning to improve memory controllers [70]. Using reinforcement learning, memory controllers could learn to optimize its scheduling policy on the fly. Jia et al. investigated memory request scheduling for massive parallel processors (GPUs) [73]. They employed request reordering and cache bypassing to prioritize memory requests. Jiang et al. studied granularity of DRAM caches [74]. They argued that DRAM caches should only cache hot memory pages. Qureshi and Loh investigated the trade-off of latency and hit rate of DRAM caches [125]. Their results showed that DRAM cache should optimize for latency first. Ghose et al. investigated prioritizing critical loads with processor hints [55]. They monitored the loads that stalled processors for the longest times, and instructed memory controllers to prioritize these loads. Mukundan and Martinez proposed a reconfigurable MC for different optimization purposes [108]. They designed configurable memory controllers to serve three different goals, performance, energy efficiency and fairness. Herrero et al. proposed per-thread row buffers to improve DRAM hit ratio [63]. Noticing that co-running threads contended for DRAM row buffers, the authors proposed to add row buffers dedicated to each thread to mitigate this contention. Islam and Stenstrom designed a strategy to eliminate unnecessary memory accesses [71]. They proposed new architecture design to eliminate various unnecessary loads, such as forwarded loads and zero-value loads. Mutlu et al. proposed improvements to memory controllers on multi-core platforms [110, 111]. Their improved memory controllers scheduled memory requests to ensure the fairness among co-running threads. Zhang et al. studied the problem of varied write recovery time in future DRAM chips [167]. They proposed to employ chunk-specific write recovery control and chunk remapping to exploit performance benefits of fast chunks. Chatterjee et al. proposed a technique called "Staged Reads" to process reads and writes simultaneously [27]. "Staged Reads" issued reads to idles banks that were not processing writes to parallelize the processing of reads and writes. Du et al. proposed a hardware-software co-design technique to allocate superpages in physical memory with retired physical pages [40]. Retried physical pages put holes in continues physical address space and prevents the allocation of superpages. The authors designed new page table formats which allow the allocation of

superpages even when retired physical pages exist. Stuecheli et al. reduced the number of write-to-read switches with virtual write queues [141]. They proposed a method to coordinate the cache write-backs with memory controllers. More specifically, they let memory controllers initiate write-backs to group writes and reduce write-to-read switches. Unlike these hardware techniques, our models and run-time systems do not require any hardware modification. Therefore, our models and run-time systems work on existing platforms. Additionally, these hardware techniques mainly affect DRAM timing, DRAM row buffer hit ratios and memory bandwidth utilization. Our memory models can be revised to consider these new hardware designs by updating their probability equations or constraints accordingly.

# Chapter 3

# Memory Bandwidth Limitation on Large-scale NUMA Platforms

This chapter provides a detailed analysis on the memory bandwidth limitation on NUMA platforms. We begin this chapter with a thorough description of NUMA architectures. We then discuss how memory bandwidth is a key factor in determining optimal core allocations. This chapter also discusses the major technical challenges that must be addressed to predict optimal core allocations on large scale NUMA machines.

## 3.1 The NUMA Architecture

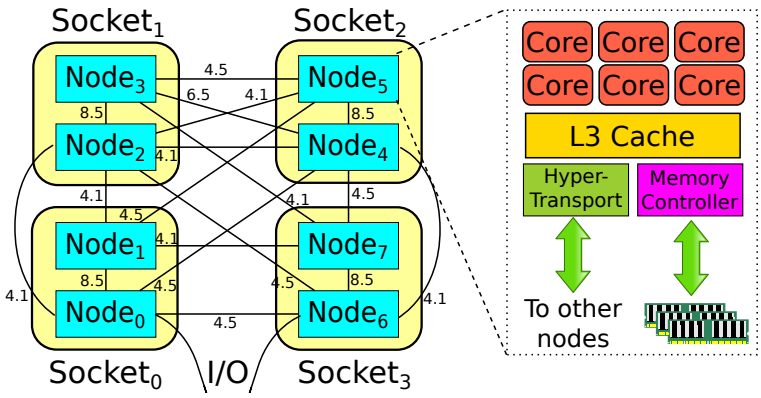



Figure 3.1: A NUMA machine with AMD Opteron 6174 processors. This machine has eight nodes on four 12-core processors. The numbers on the inter-node links represents their maximum bandwidth (GB/s).

With the ever growing need for speed, modern computer platforms are required to provide dozens or hundreds of cores to continuously provide better performance. However, because of the huge speed discrepancy between CPU and DRAM, a single DRAM connection can never provide enough memory bandwidth to satisfy the need of large numbers of cores. As a result, systems with large numbers of cores are designed as Non-uniform Memory Access (NUMA) architectures, which provide multiple DRAM connections to increase maximum memory bandwidth.

More specifically, large-scale NUMA platforms are composed of several multi-core processors that are attached to different sockets. A multi-core processor is usually composed of one or more groups of cores, which are called nodes. A node is configured to connect to its own set of DRAM modules. This configuration allows the simultaneous access of multiple DRAM modules which increases the aggregate memory bandwidth. The nodes are also connected using high-speed inter-node connections, allowing one processor to access the DRAM modules of another node [66, 148].

Figure 3.1 gives an example of a NUMA machine with four AMD Opteron 6174 processors on four sockets [6]. Each processor consists of two nodes of six cores. In total, there are 48 cores. Each node is connected to its own memory controller and DRAM modules.

The lines in Figure 3.1 also illustrate the inter-node connections (HyperTransport links [148]). Note that, because the number of inter-node connection ports per node is limited, not all nodes are directly connected, and the connection topology is not uniform. Additionally, the inter-node connections are heterogeneous in that they have different bit widths, and thus different maximum bandwidth.

Figure 3.1 also shows the maximum inter-node memory bandwidth (GB/s) of the inter-node connections. There are two reasons for the differences in the maximum bandwidth between nodes. First, the inter-node connection between two nodes within one processor is always faster than the connection of two nodes on two different processors. Second, different inter-node connections have different bit widths. An AMD Opteron 6174 node has only two 32-bit HyperTransport ports. In order for a node to connect to more than four other nodes, one 32-bit port can be partitioned into several 16-bit or 8-bit links. Obviously, a connection with wider connection (i.e., more bits) offers more bandwidth than a narrower connection.

Figure 3.2 gives another example of a NUMA machine. This machine has four Intel Xeon X7550 processors, where each processor has one node of eight cores. Again, the lines and the numbers in Figure 3.2 also show the connection topology and maximum

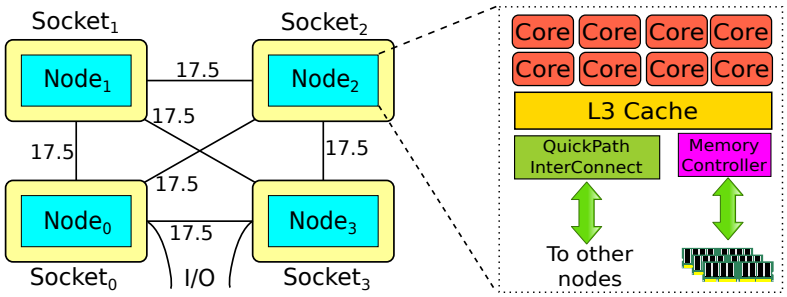



Figure 3.2: A NUMA machine with Intel Xeon X7550 processors. This machine has four nodes on four 8-core processors. The numbers on the inter-node links represents their maximum bandwidth (GB/s).

bandwidth of the inter-node connections.

On the Intel platform, the nodes are connected with QuickPath Interconnect (QPI) [66]. Unlike the AMD NUMA machine in Figure 3.1, this Intel NUMA machine has fewer nodes. Therefore, there are enough QPI ports so that all nodes are directly connected, and the inter-node connections have the same maximum bandwidth.

## 3.2 Memory Bandwidth Impact on Core Allocation

To understand how memory bandwidth limitation impacts the optimal core allocations on large-scale NUMA machines, we conducted a series of experiments using PARSEC, NPB and BLAS benchmarks on the two NUMA machines described previously in Section 3.1 [7, 15, 75]. Our experimental results show that there are three memory bandwidth factors that impact core allocation and limit scalability. This section discusses these three factors in detail. In this dissertation, a core allocation is described using a vector $\{a_0, a_1, a_2, \cdots, a_i, \cdots, a_n\}$, where $a_i$ represents the number of cores allocated on node $i$.

### 3.2.1 Factor 1: Local Memory Bandwidth Limitation

Local memory bandwidth usage refers to the memory bandwidth that is consumed by application threads when they access the DRAM modules that are directly connected to the cores on which these threads are executing. Depending on the memory bandwidth demand, the local memory bandwidth can limit core allocation.

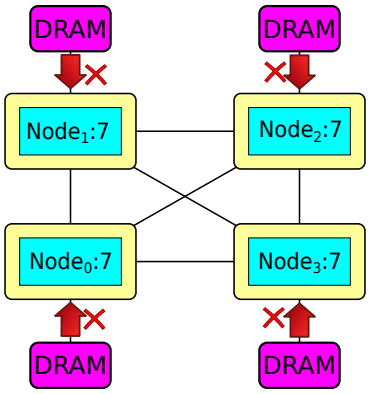



Figure 3.3: Benchmark *mg.D* is limited by local memory bandwidth on the Intel NUMA machine. The number on each node represents the optimal number of cores should be allocated on that node. A "cross" indicates a saturated memory connection.

For example, the core allocation for NPB benchmark *mg.D* on the Intel NUMA machine is limited by local memory bandwidth when it is in the phase of executing function *resid*. The optimal core allocation for *mg.D* and the saturated memory connections are illustrated in Figure 3.3. As the figure shows, the optimal core allocation for *mg.D* is $\{7, 7, 7, 7\}$, i.e., seven cores per node. *Mg.D* has only local memory accesses when executing the function *resid*. Because the local DRAM modules cannot provide enough bandwidth, only seven cores (out of eight) have to be allocated to achieve the best performance. The optimal core allocation of $\{7, 7, 7, 7\}$ performs 10% faster than the use-all-cores allocation (using all eight cores on all four nodes).

### 3.2.2 Factor 2: Inter-node Memory Bandwidth Limitation

Inter-node memory bandwidth usage refers to the usage that is consumed by application threads when they communicate with threads running on other nodes to collaborate on a task. The inter-node connections usually have limited maximum memory bandwidth. If the data demand for inter-node communication is high, then the inter-node connections may not be able to provide enough bandwidth to satisfy the need of all cores.

Fore example, the NPB benchmark *mg.D* requires many inter-node communications when it is executing the function *rprj3*. In this function, *mg.D*'s threads communicate in a ring fashion as illustrated in Figure 3.4, i.e., the threads on $node_0$ send data to $node_1$, the threads on $node_1$ send data to $node_2$, etc.

Recall that in Section 3.1, we discussed that different inter-node connections have

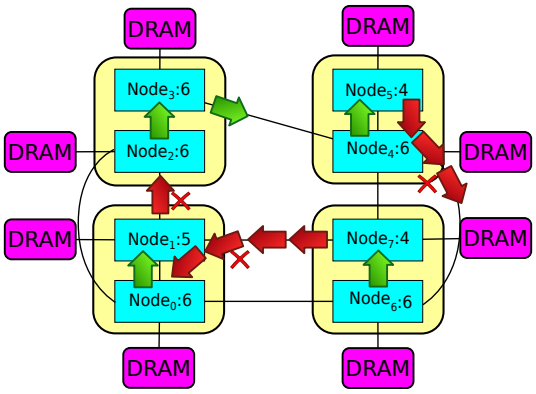



Figure 3.4: NPB benchmark *mg.D* is limited by inter-node memory bandwidth on the AMD NUMA machine when executing the function *rprj3*. The number on each node represents the optimal number of cores that should be allocated on each node. A "cross" indicates a saturated memory connection. Note that the amount of local data usage is limited. However, for clearer illustration, local data accesses and unused inter-node connections are omitted.

different maximum bandwidths. Because of this difference, nodes connected to different inter-node connections have different core allocations. More specifically, $node_0$, $node_2$, $node_4$, and $node_6$ can only send data through high-bandwidth inter-node links which are within processors. Therefore, all cores on these nodes can be allocated. Because the connection between $node_1$ and $node_2$ is cross-processor and has limited bandwidth, only five cores can be allocated on $node_1$. $Node_3$ also sends data using cross-processor inter-node connection to $node_4$. However, because the connection between $node_3$ and $node_4$ has high bit-width, this connection also has high bandwidth as shown in Figure 3.1. This high bandwidth allows all six cores of $node_3$ to be used. For $node_5$ and $node_7$, not only do they send data cross-processor, but they also use multiple hops of connections to send data. Consequently, only fours cores on each of $node_5$ and $node_7$ should be allocated. In summary, the optimal core allocation for this case is $\{6, 5, 6, 6, 6, 4, 6, 4\}$ on the AMD platform, which is 9% faster than use-all-cores allocations. It also worth noting that, because the inter-node connections on the Intel platform have higher bandwidth, the same benchmark performs the best with use-all-cores allocation on the Intel platform.

### 3.2.3 Factor 3: Interference of Local and Inter-node Memory Accesses

Not only do local memory accesses and inter-node memory accesses impact core allocations independently, they may also interfere with each other. This interference can also impact core allocations. Because this interference varies with the locations of data, there are two interference cases that have to be considered.

#### Case 1: Fully Shared Data

For some multi-threaded applications, the data are shared by all threads, and all of the data are held in the DRAM modules of one single node. Without loss of generality, let the node with shared-data be $node_i$. Because all of the data are located on $node_i$'s DRAM modules, these DRAM modules have to perform two tasks: they must send data to the cores on $node_i$ to satisfy these cores' computation needs, and they must also send data to the cores on the other nodes to satisfy their needs. However, if the data demand is high, the maximum output bandwidth of $node_i$'s DRAM may not be large enough to satisfy the needs of all cores. As a result, some of the cores cannot be allocated. In this case, because all data are located on $node_i$, it is better to satisfy the needs of $node_i$'s cores first. After the needs of $node_i$'s cores are met, we can use the remaining bandwidth to satisfy the needs of some cores on the other nodes.

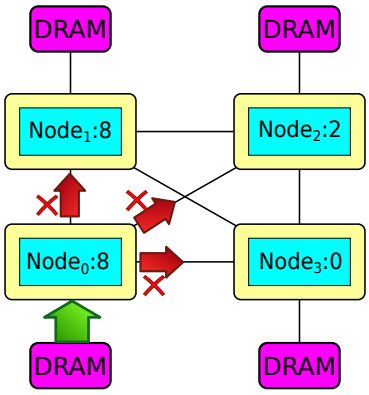



Figure 3.5: Benchmark *streamcluster* is limited by inter-node memory bandwidth on the Intel NUMA machine. The number on each node represents the optimal number of cores that should be allocated on that node. A "cross" indicates a saturated memory connection.

For example, the PARSEC benchmark *streamclsuter* is an application with this all-data-shared behavior. Figure 3.5 gives the optimal core allocation for *stream-*

*cluster* and the saturated memory connections when it executes on the Intel NUMA machine. As the figure shows, the maximum output memory bandwidth of $node_0$, which contains shared data, cannot satisfy the needs of all cores on all nodes. After meeting the needs of all eight cores on $node_0$, the remaining memory bandwidth can only support the execution of eight cores on $node_1$ and two cores on $node_2$. Therefore, the optimal core allocation for *streamcluster* on the Intel NUMA platform is $\{8, 8, 2, 0\}$, which is 79% faster than the use-all-cores allocation.

**Case 2: Partially Shared Data**

There are also applications where the majority of their data are local to their threads, while only a small portion of the data is shared by all threads. That is, most data are distributed among all nodes, and these data are only accessed by the threads running on these data's local (home) nodes, while some data are held in one node for sharing. Without loss of generality, let the node with shared-data be $node_i$. Because $node_i$'s DRAM modules have both local data and shared data, these DRAM modules have to satisfy two data needs: they must send data to the cores on $node_i$ to satisfy these cores' needs for the local data, and they must also send data to the cores on the other nodes to satisfy their needs for the shared data. However, if the data demand is high, the maximum output bandwidth of $node_i$'s DRAM may not be large enough to satisfy both needs. As a result, some of the cores cannot be allocated. Unlike case 1, in this case (case 2), because most data are located on the nodes other than $node_i$, it is better to satisfy the needs of the other nodes' cores first. After the needs of the other node's cores are met, we can use the remaining bandwidth to satisfy the needs of some cores on $node_i$.

For example, the BLAS routine *dgemm* implemented by AMD Core Math Library (ACML) is an application with this partial-data-sharing behavior [7]. Figure 3.6 gives the optimal core allocation for *dgemm* and the saturated memory connections on the AMD NUMA machine. As the figure shows, the threads of *dgemm* access local memory, as well as the shared memory on $node_0$. Because of the high data demand, the DRAM connection of $node_0$ cannot provide enough bandwidth for both local-data and shared-data requests. To reduce this contention, either some cores on $node_0$ or some cores on the other nodes should not be allocated. Because most data are local, every thread has more local-data requests than shared-data requests. Consequently, reducing the core allocations on the other nodes (nodes other than $node_0$) does not significantly reduce shared-data requests, unless significant numbers of cores on these

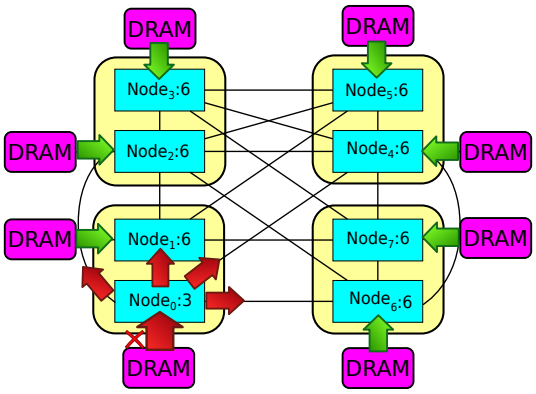



Figure 3.6: BLAS routine *dgemm* is limited by inter-node memory bandwidth on the AMD NUMA machine. The number on each node represents the optimal number of cores that should be allocated on that node. A "cross" indicates a saturated memory connection.

nodes are not used, which slows down the processing of the local data on these nodes and hurts performance. On the other hand, not-allocating a few cores on $node_0$ can significantly reduce the local requests on $node_0$, and thus significantly mitigate the local/shared-data contention, without hurting performance. That is, the best core allocations for partial-data-shared applications are to limit the core allocation on the shared data node. For this specific application of *dgemm*, the optimal core allocation is $\{3, 6, 6, 6, 6, 6, 6, 6\}$, which is 12% faster than use-all-cores allocation.

## 3.3 Summary of Insights

Based on the hardware configurations in Section 3.1 and the examples in Section 3.2, the following insights can be summarized:

1. Both local DRAM connections and inter-node connections can affect optimal core allocations. Therefore, both local bandwidth usage and inter-node bandwidth usage have to be predicted and considered for optimal core allocation prediction.

2. Local and inter-node memory accesses can interfere with each other. This interference limits memory bandwidth usage and impacts core allocations. Therefore, the local and inter-node contention must be accurately estimated for bandwidth usage and optimal core allocation prediction.

3. Application communication patterns and memory behaviors also greatly affect memory bandwidth usage and optimal core allocations. As a result, the predictions for bandwidth usages and optimal core allocations should be made based on real application behavior observed at run-time.

4. Hardware configuration is another determining factor of memory bandwidth usages and optimal core allocations. Consequently, the predictions for bandwidth usage and core allocations should be made based on the actual hardware configuration.

5. The inter-node connections on a NUMA machine can be heterogeneous, and the connection topology may be non-uniform. This heterogeneity and non-uniformity dictates that different nodes may have different numbers of cores allocated in an optimal core allocation. Hence, each inter-node connection and each node must be treated differently when predicting optimal core allocations.

Armed with the above insights, this research focused on the prediction of local and inter-node memory bandwidth, the run-time prediction of optimal core allocation for each node, and the run-time adaption to optimal core allocations.

# Chapter 4

# Predicting the Local Memory Bandwidth Usages

## 4.1 Introduction

As discussed in Chapter 3, the local memory bandwidth alone can limit the scalability of multi-threaded applications. Therefore, for applications that are limited by local memory bandwidth, the prediction of their optimal core allocations requires the prediction of their local memory bandwidth usages.

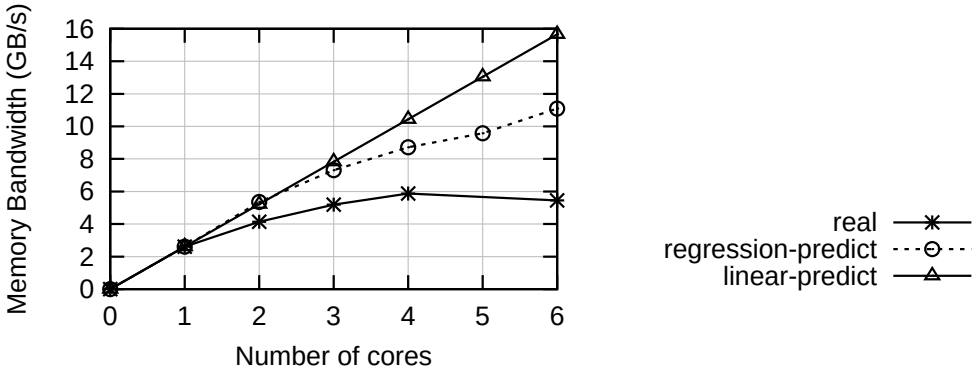



Figure 4.1: Memory bandwidth usage of *facesim* on one node of an AMD Opteron 6174 processor with linear prediction model and a regression-based model.

Unfortunately, the memory bandwidth models used in previous research have low accuracy. They rely on simple mathematical models or regression analysis, and only consider samples of bandwidth usage for prediction. DRAM contention and DRAM concurrency, as well as other important factors (e.g., program memory behaviors) are

overlooked. Figure 4.1 gives the bandwidth usage of a PARSEC benchmark *facesim* running on an AMD 6-core node [15], as well as the bandwidth predictions of two popular models. One is a linear model that assumes memory bandwidth usages increase linearly with the number of cores [86, 136, 144]. The other is based on multiple logarithmic and linear regressions [81]. As the figure shows, both models have low accuracy. The linear model has an average accuracy of 14% while the regression-based model has an average accuracy of 44% in this example.

To the best of our knowledge, no existing model can provide highly accurate local memory bandwidth usage predictions on real machines, because of four major challenges,

- The first challenge is to accurately predict the contention for DRAM resources from co-running threads. The severity of DRAM contention varies with program behavior in a complicated manner. However, because DRAM contention significantly impacts bandwidth usage, it must be correctly predicted.

- The second challenge is to accurately predict the DRAM concurrency. DRAM requests accessing different banks can be served simultaneously and overlap with each other. This overlapping further complicates the prediction of the latency of DRAM requests.

- The third challenge is to consider the large variety of hardware and software factors that affect bandwidth usage besides contention and concurrency. These factors have to be clearly identified and carefully considered.

- The last challenge is to develop a model that can be computed in a short amount of time. Some uses (e.g., resource contention management) require a fast model that can be applied during execution to handle dynamic workloads without incurring too much overhead.

To address these challenges, this chapter presents DraMon, a highly accurate bandwidth model that considers a wide range of factors. We demonstrate that predicting bandwidth usages requires predicting DRAM contention (e.g., row buffer hit ratio) and DRAM concurrency. We experimentally show that contention and concurrency can be predicted with high accuracy and with low time overhead using probability theory. We developed two versions of DraMon: DraMon-T, an offline memory-trace based model, and DraMon-R, a run-time model which uses performance monitoring units (PMUs) as inputs.

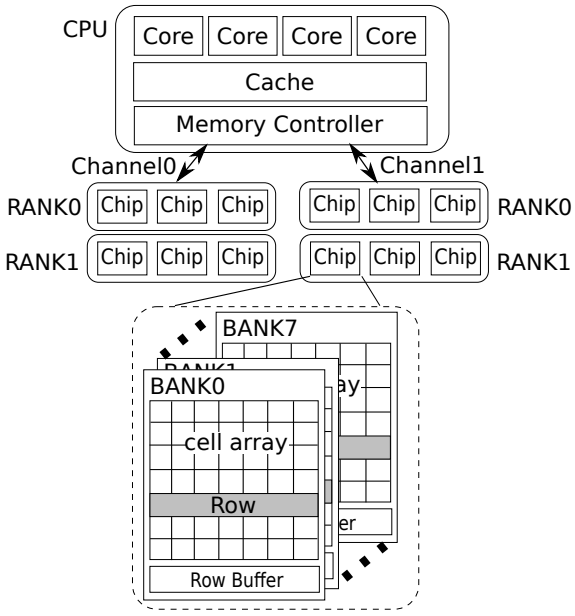



Figure 4.2: Memory/DRAM systems.

The rest of the chapter is organized as follows. Section 4.2 discusses DRAM systems. Section 4.3 provides a high level overview of DraMon. Section 4.4 explains DraMon in detail. Section 4.5 describes how to obtain input parameters. Section 4.6 evaluates DraMon on a real machine. Section 4.7 discusses other related issues. Section 4.8 summarizes this chapter.

## 4.2 Memory System Background

Before introducing DraMon, this section describes the memory systems on contemporary multi-core platforms.

### 4.2.1 DRAM Architecture

Figure 4.2 depicts a generic memory system and DRAM structure. The on-chip memory controller (MC) is connected to several **channels**. A DRAM request can access one channel at a time, or it can access all channels at once, depending on the configurations of the MC. A channel is composed of several **ranks**. A rank can be roughly viewed as a memory module. Each rank has several memory chips. A memory chip is composed of several **banks**. Each bank is essentially a cell array where a cell is used to store 0 or 1. The banks with the same index on all chips form

a conceptual bank of a rank. For example in Figure 4.2, the $BANK0$s of all chips of $RANK1$ form its conceptual $BANK0$. When $BANK0$ is accessed, the $BANK0$s of every chip on $RANK1$ are activated simultaneously.

### 4.2.2 DRAM Request Types

Each bank (both conceptual and physical) has a row buffer. When accessing a piece of data, the row containing this data is read into the row buffer. Then the target data is extracted and sent to MC. Figure 4.3 shows the operations of a DRAM read [72]. First, the connections between the row buffer and the cell array are precharged (PRE). This precharge is crucial for stable reading, and it requires **tRP** time. Next, the MC issues a row access (RA) command and reads the row into the row buffer in **tRCD** time. After the row is ready, MC sends the column address (CA) and locates the target data in **tCAS** time. Finally, the data is extracted and sent to MC using **tBurst** time.

Depending on the status of the target bank, a DRAM request falls into one of three categories [6]:

- **Hit**: The row buffer has the desired row. Only column access and data transportation is required. Therefore, the latency of a hit is $tCAS + tBurst$.
- **Miss**: The bank is precharged, but the row buffer is empty. The desired row has to be read into the row buffer. Therefore, the latency of a miss is $tRCD + tCAS + tBurst$ .
- **Conflict**: The bank is not yet precharged for this request. A precharge is required. Therefore, the latency of a conflict is $tRP + tRCD + tCAS + tBurst$.

Note that although a conflict can be viewed as a miss, its service time is very different. This difference is important for accurately predicting average DRAM request latency and bandwidth usage.

### 4.2.3 DRAM Contention

If we compare a row buffer with a cache line, we can easily see similarities. They are both used to temporarily store a copy of data. They are both rewritten to store active data. And they are both shared by, and contended for, by co-running threads. Consequently, similar to predicting cache contention, which is to predict the hit/miss

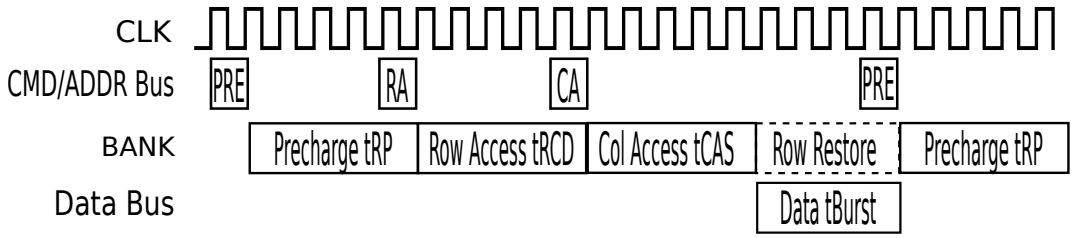



Figure 4.3: A read cycle.

ratios, predicting DRAM contention is essentially predicting the ratios (percentages) of the three types of DRAM requests: $Ratio_{hit}$, $Ratio_{miss}$ and $Ratio_{conf}$.

### 4.2.4 DRAM Concurrency

The latencies discussed in Section 4.2.2 are single-request latencies. In practice, multiple DRAM requests can be served simultaneously, which greatly reduces their average latency. Figure 4.4 shows four consecutive hits, assuming both $tCAS$ and $tBurst$ are 4 cycles [72]. As the figure shows, the column open operation ($tCAS$) can overlap with data transportation ($tBurst$). Therefore, it takes 21 cycles to serve all four requests. Thus, the average latency of these hits is $\frac{21}{4} = 5.25$ cycles, while the full single-request hit latency is $tCAS + tBurst = 8$ cycles.

Similarly, the operations of miss and conflict can also overlap with other DRAM requests. Because this concurrency can significantly reduce average DRAM latency, it must be carefully considered in a bandwidth model.

### 4.2.5 Memory Controller Optimizations

Because of the large differences between the latencies of hit, miss and conflict, memory controllers employ several optimization techniques. Accurately predicting memory bandwidth requires considering these optimizations.

The first common optimization is a closed or adaptive page policy, where an opened row buffer is automatically closed and precharged if it is idle for some time [72]. Later, a request to the same bank can then proceed to open a new row without having to wait for precharging. *Four Bank Activation Window (FAW)* also increases the chance of automatic row buffer closing [72]. Because of power constraints, only four banks can be activated in a rank within a certain time window. This relatively long time window renders opened banks idle and more likely to be automatically closed.

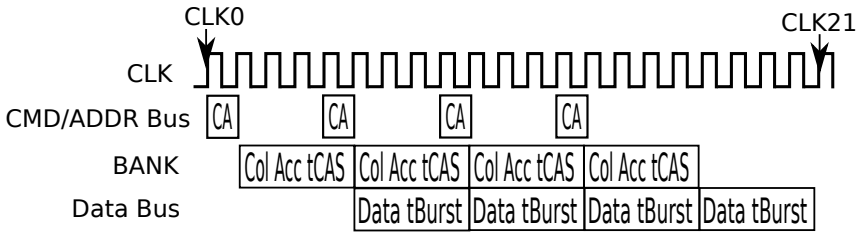



Figure 4.4: Four concurrent DRAM hits.

The second common optimization is request reordering [72]. If a request in the MC queue hits an open row, the MC may issue it before the requests that are queued earlier, to take advantage of the low latency of DRAM hits.

## 4.3 Overview of DraMon Model

As stated previously, accurately predicting bandwidth usage requires considering DRAM contention and DRAM concurrency. This section theoretically connects DRAM contention and DRAM concurrency to bandwidth usage, and serves as a road map for the following sections.

### 4.3.1 Predicting Bandwidth from DRAM Contention and Concurrency

Memory bandwidth ($BW$) usage is basically the product of the number of channels, $chnl\_cnt$, available, the memory request rate, $Rate_{mem}$, per channel (the number of requests finished per second), and the size of each request, $Size_{mem}$:

$$BW = chnl\_cnt \times Rate_{mem} \times Size_{mem}. \tag{4.1}$$

Memory request rate is determined by two factors: (1) $Rate_{issue}$, the maximum issue rate of DRAM requests limited by program behavior, and (2) $Rate_{dram}$, the DRAM service rate limited by DRAM contention. The actual memory request rate is limited by the smaller of the two:

$$Rate_{mem} = min(Rate_{issue}, Rate_{dram}). \tag{4.2}$$

Therefore, predicting memory request rate is reduced to the problem of predicting issue rate and DRAM service rate.

Predicting DRAM service rate is equivalent to predicting the reciprocal of the average DRAM request latency $Lat_{dram}$.[1] DRAM latency can be further divided into two components, the average read request latency ($Lat_r$) and the average write request latency ($Lat_w$). As an optimization, modern MCs delay write requests and group them together for issuing [27, 72]. Because reads and writes are processed separately by MCs, their average latencies can be computed separately. Their weighted average is then the average DRAM latency. Additionally, switching from writes to reads requires stalling the data bus, which adds an extra overhead ($O_{wtr}$). Similarly, when multiple ranks are accessed, rank-to-rank switching also requires data bus stalls and adds overhead ($O_{rtr}$). Assume the read request ratio is $Ratio_r$ and the write request ratio is $Ratio_w$. Summarizing the above, we have:

$$\begin{aligned} Rate_{dram} &= \frac{1}{Lat_{dram}}, \\ Lat_{dram} &= Ratio_r \times Lat_r + Ratio_w \times Lat_w + O_{wtr} + O_{rtr}. \end{aligned} \tag{4.3}$$

Similar to predicting the average cache latency, the average read/write latency can be computed using the following equations [62]:

$$\begin{aligned} Lat_r =& Ratio_{hit} \times Lat_{r,hit} + Ratio_{miss} \times Lat_{r,miss} + \\ & Ratio_{conf} \times Lat_{r,conf} \\ Lat_w =& Ratio_{hit} \times Lat_{w,hit} + Ratio_{miss} \times Lat_{w,miss} + \\ & Ratio_{conf} \times Lat_{w,conf}. \end{aligned} \tag{4.4}$$

Note that the hit/miss/conflict (HMC) ratios here are the average ratios of both reads and writes. The actual ratios of reads and writes are different. However, because their latencies are close (differ by one cycle), using average HMC ratios for both reads and writes is a an acceptable approximation.

[1] The DRAM latency here is not the single-request latency. It is rather the average latency of multiple overlapped requests, which is the time between they are issued from the MC and the data returned to the MC. It is only used to predict memory bandwidth.

Combining the above equations gives the following equation,

$$\begin{aligned}\mathbf{BW} =& chnl\_cnt \times min(\mathbf{Rate_{issue}}, \frac{1}{Ratio_r \times Lat_r + Ratio_w \times Lat_w + \mathbf{O_{wtr}} + \mathbf{O_{rtr}}}) \\ & \times Size_{mem}, \\ Lat_x =& \sum_{ty} \mathbf{Ratio_{ty}} \times \mathbf{Lat_{x,ty}}, \quad \text{where } x \in \{r, w\},\ ty \in \{hit, miss, conf\}.\end{aligned} \tag{4.5}$$

This equation connects bandwidth usage to DRAM contention (HMC ratios, $Ratio_{ty}$) and DRAM concurrency (HMC latencies, $Lat_{r/w,ty}$). Predicting bandwidth also requires predicting the maximum issue rate $Rate_{issue}$, the write-to-read switching overhead $O_{wtr}$ and the rank-to-rank switching overhead $O_{rtr}$. Four variables ($chn\_cnl$, $Size_{mem}$, $Ratio_r$, $Ratio_w$) are acquired from memory traces or PMUs.

### 4.3.2 Model Algorithm

**Algorithm 1** Algorithm of DraMon

```
collect hardware related parameters (Sect. 4.5.2)
for all program p do
    collect software related parameters (Sect. 4.5.3)
    for all core/thread count n do
        predict maximum issue rate Rate_issue (Sect. 4.4.1)
        predict HMC ratios Ratio_ty (Sect. 4.4.2)
        predict HMC latencies Lat_r/w,ty (Sect. 4.4.3)
        predict write-to-read overhead O_wtr (Sect. 4.4.4)
        predict rank-to-rank overhead O_rtr (Sect. 4.4.5)
        predict bandwidth usage BW (Eq. (4.5))
    end for
end for
```

The steps of using DraMon are explained in Algorithm 1, which requires several parameters as inputs, including parameters that describe a platform's hardware configuration and the parameters that describe a program's memory behavior. With these parameters, DraMon predicts the memory bandwidth for a multi-threaded program running with a certain number of cores/threads using equations (4.5).

## 4.4 The Bandwidth Model in Detail

This section discusses the prediction of the unknown components of equation (4.5). Note that our prediction equations require several input parameters. Obtaining these parameters is discussed in Section 4.5.

### 4.4.1 Predicting Issue Rate

For a multi-threaded program, if the issue rate of a single core/thread is $Rate_{issue\_single}$ (input parameter obtained in Section 4.5), then its maximum possible issue rate when running with $n$ cores/threads is

$$Rate_{issue} = Rate_{issue\_single} \times n. \tag{4.6}$$

### 4.4.2 Predicting HMC ratios

Here we describe the prediction of the hit/miss/conflict (HMC) ratios of one thread of a multi-threaded program. The same process can be applied to predict the HMC ratios of others threads. The overall HMC ratios of the program is then the average of all threads' HMC ratios.

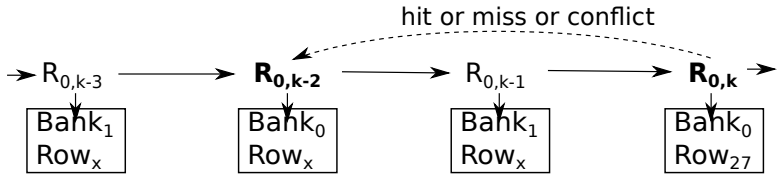
We first provide an example with two threads to illustrate the basic idea of our DRAM contention prediction. Then we expand this idea to handle any number of threads and describe the equations for predicting HMC ratios. For simplicity, we focus on predicting hit ratio first.

#### A Two-thread Example

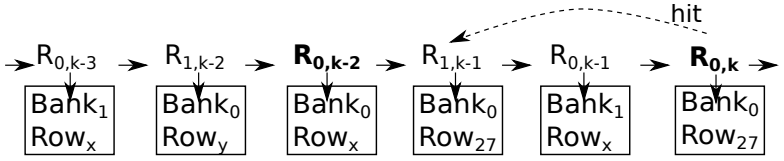
Consider a case where two threads $T_0$ and $T_1$ are executing simultaneously. First we predict the $T_0$'s hit ratio. Naturally, predicting the hit ratio requires identifying the hits in $T_0$'s requests and computing their percentage. However, this approach requires processing millions of requests, which is time consuming. Here we use a key insight into the relationship between hit ratio and hit probability.

*Insight 1: The hit ratio of $T_0$ is equivalent to the probability that one of its requests is a hit. Conversely, predicting the hit ratio of $T_0$ is equivalent to predicting the probability that an arbitrary request of $T_0$ is a hit.*

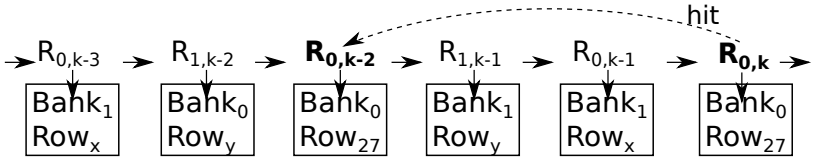
This insight allows us to focus on one single request. It also permits predicting the hit ratio using probability theory which greatly reduces prediction time. Without




(a) Requests when there is only $T_0$ running.




(b) Case 1: Request $R_{0,k}$ hits the row opened by $R_{1,k-1}$.




(c) Case 2: Request $R_{0,k}$ hits the row opened by $R_{0,k-2}$.

Figure 4.5: Predicting the hit ratio of thread $T_0$ when it is running with another thread $T_1$.

loss of generality, here we predict the hit probability of the $k$'th request of $T_0$, denoted by $R_{0,k}$.

Whether $R_{0,k}$ is a hit, depends on its preceding requests. Figure 4.5a gives a sequence of requests when $T_0$ runs alone. The box under each request gives its bank and row addresses. In this figure, $R_{0,k-2}$ is the last request before $R_{0,k}$ that accesses the same bank ($Bank_0$) used by $R_{0,k}$. Depending on $R_{0,k-2}$'s row address, $R_{0,k}$ can be a hit, miss or conflict. If $R_{0,k-2}$ also accesses $Row_{27}$, $R_{0,k}$ is a hit. If $R_{0,k-2}$ accesses a row other than $Row_{27}$, $R_{0,k}$ is a conflict. If the row opened by $R_{0,k-2}$ is closed by the MC, then $R_{0,k}$ is a miss.

Similarly, when there is co-running thread $T_1$, the type (hit/miss/conflict) of $R_{0,k}$ depends on its preceding requests from both threads. However, there are billions of requests preceding $R_{0,k}$, and it is impossible to consider all of them. Here, we gain our second key insight from Figure 4.5.

*Insight 2: Only requests issued after $R_{0,k-2}$ (including $R_{0,k-2}$) have to be considered when predicting the type of $R_{0,k}$, because any change made to $Bank_0$'s row buffer by requests before $R_{0,k-2}$ is reset by $R_{0,k-2}$.*

This insight greatly reduces the number of preceding requests that require consideration. Figure 4.5b and 4.5c show the requests from both $T_0$ and $T_1$. In these two

figures, only one of $T_1$'s requests, $R_{1,k-1}$, is issued between $R_{0,k-2}$ and $R_{0,k}$, and its destination affects the type of $R_{0,k}$.

There are two cases where $R_{0,k}$ is a hit. In case 1 (Figure 4.5b), $R_{1,k-1}$ accesses the same row ($Row_{27}$) used by $R_{0,k}$. Therefore, $R_{0,k}$ hits the row opened by $R_{1,k-1}$. In case 2 (Figure 4.5c), $R_{1,k-1}$ does not access the same row ($Row_{27}$) used by $R_{0,k}$. However, $R_{0,k-2}$ accesses $Row_{27}$. Therefore, $R_{0,k}$ hits the row opened by $R_{0,k-2}$. If the probabilities of case 1 and case 2 are $P_1$ and $P_2$, then the probability that $R_{0,k}$ is a hit is $P_1 + P_2$, which is also the hit ratio of $T_0$.

In summary, the two key insights above allow us to focus on one request and a limited number of its preceding requests. By enumerating the destinations of the preceding requests, we list all the cases which can produce a hit. Then we predict the probabilities of these cases, the sum of which is then the hit ratio. Next we generalize this example to any number of threads, and predict the case probabilities.

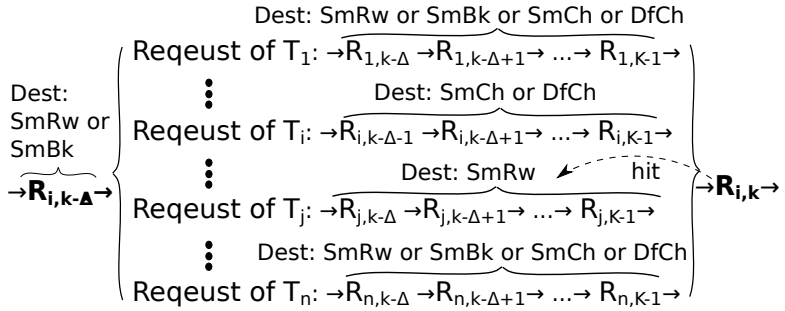
### Generalizing to $n$ threads

Consider the case where $n$ threads are running simultaneously. Here we predict the hit ratio of the $i$'th thread $T_i$. According to Insight 1, predicting $T_i$'s hit ratio is equivalent to predicting the probability that its $k$'th request $R_{i,k}$ is a hit.

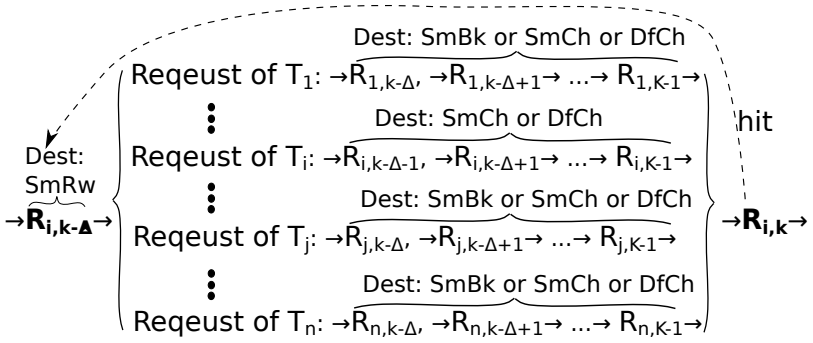
Assume the nearest preceding request from $T_i$ that accesses the same bank used by $R_{i,k}$ is $R_{i,k-\Delta}$. Insight 2 can be generalized as: only requests after $R_{i,k-\Delta}$ (including $R_{i,k-\Delta}$) should be considered when determining the type of $R_{i,k}$. Moreover, we define the **Bank Reuse Distance (BRD)** of $R_{i,k}$ as $\Delta$. In Figure 4.5, $R_{0,k}$'s BRD is 2 because the nearest same-bank accessing request from $T_0$ is $R_{0,k-2}$.

Figure 4.6a gives the sequence of requests issued between $R_{i,k-\Delta}$ and $R_{i,k}$ by all $n$ threads. Each row between $R_{i,k-\Delta}$ and $R_{i,k}$ represents the requests issued from one thread. Threads are depicted in independent rows because they execute in parallel. The type (hit/miss/conflict) of $R_{i,k}$ depends on preceding requests' destinations, which fall into four categories:

- $SmRw$, the same row used by $R_{i,k}$.
- $SmBk$, a different row on the bank used by $R_{i,k}$.
- $SmCh$, a different bank of the channel used by $R_{i,k}$.
- $DfCh$, a different channel than $R_{i,k}$'s channel.




(a) Case 1: Request $R_{i,k}$ is a hit when $T_j$'s requests access the same row used by $R_{i,k}$.




(b) Case 2: Request $R_{i,k}$ is a hit when $R_{i,k-\Delta}$ accesses the same row used by $R_{i,k}$.

Figure 4.6: Predicting the hit ratio of thread $T_i$ when it is running with $n-1$ threads.

To reduce computation time, we assume that all requests from the same thread have the same destination, i.e., the same row. Because of data locality, this assumption holds for most programs (more than 85% consecutive requests of our benchmarks hit the same row). In Figure 4.6, the destination of one thread is marked above its requests.

Similar to the example in Figure 4.5, there are two cases where $R_{i,k}$ is a hit. In case 1 (Figure 4.6a), at least one thread $T_j$ has a destination of $SmRw$, and $R_{i,k}$ hits the row opened by $T_j$'s requests. In case 2 (Figure 4.6b) none of the middle threads accessing $SmRw$. However, $R_{i,k-\Delta}$ accesses this row, and $R_{i,k}$ hits the row opened by $R_{i,k-\Delta}$.

The hit probability of $R_{i,k}$, which is also the hit ratio of $T_i$, is the sum of the probabilities of these two cases. Additionally, the total number of preceding requests have to be determined. Too many preceding requests may cause the MC to close the bank accessed by $R_{i,k}$ (recall the adaptive page policy). Next we discuss predicting these values.

### Predicting the Number of Preceding Requests

From BRD's definition, there are $\Delta$ requests of $T_i$ that should be considered when predicting the type of $R_{i,k}$.

For a co-running thread $T_j$, the number of its requests to be considered depends on its issue rate. Assume the single thread issue rate of $T_i$ is $Rate_{issue,i}$, and the single thread issue rate of $T_j$ is $Rate_{issue,j}$. During the time when $T_i$ issued $\Delta$ requests, the number of requests issued by $T_j$ is

$$ReqCnt_{\Delta,j} = \Delta \times \frac{Rate_{issue,j}}{Rate_{issue,i}}. \tag{4.7}$$

### Predicting Case Probabilities and the Hit Ratio

Hit ratio prediction requires the following input parameters:

- the hit ratio of $T_i$ when there are no co-running threads, $Ratio_{hit,single}$,
- the probabilities, $P_{SmRw}$, $P_{SmBk}$, $P_{SmCh}$ and $P_{DfCh}$, that a co-running thread access SmRw, SmBk, SmCh and DfCh respectively,
- the number of requests after which an opened row-buffer is automatically closed, $D_{ac}$, and,
- the total number of threads, $n$.

Obtaining these parameters is discussed in Section 4.5. With these inputs, DraMon predicts the probabilities of the two cases in Figure 4.6 and the hit ratio of $T_i$.

*Case 1*: At least one co-running thread has a destination of SmRw (Figure 4.6a). This case can be further broken down into two sub-cases.

*Sub-case A*: No co-running thread has a destination of SmBk. The probability of sub-case A is

$$\begin{aligned} P_{caseA} &= P(\exists SmRw \wedge \nexists SmBk) \\ &= P(\nexists SmBk) - P(\nexists SmRW \wedge \nexists SmBk) \\ &= (1 - P_{SmBk})^{n-1} - (P_{SmCh} + P_{DfCh})^{n-1}. \end{aligned} \tag{4.8}$$

Clearly, sub-case A is a hit:

$$Ratio_{hit,\Delta}(caseA) = P_{caseA}. \tag{4.9}$$

*Sub-case B*: At least one co-running thread has a destination of SmBk. The probability of sub-case B is

$$\begin{aligned} P_{caseB} &= P_{case1} - P_{caseA} = P(\exists SmRw) - P(caseA) \\ &= (1 - P(\nexists SmRw)) - P_{caseA} \\ &= (1 - (1 - P_{SmRw})^{n-1}) - P_{caseA}. \end{aligned} \tag{4.10}$$

Sub-case B can be a hit or a conflict depending on whether the last access before $R_{i,k}$ is a SmRW or SmBk. Because the orders of the requests are random, the type of the last request follows uniform distribution. Therefore, approximately half of the permutations are hits:

$$Ratio_{hit,\Delta}(caseB) = \frac{1}{2} P_{caseB}. \tag{4.11}$$

*Case 2*: No co-running thread has a destination of SmRw. However, if $R_{i,k-\Delta}$ is SmRw, $R_{i,k}$ may still be a hit (Figure 4.6b). $R_{i,k-\Delta}$ is SmRw means $R_{i,k}$ is hit when there are no co-running threads. Therefore, the probability that $R_{i,k-\Delta}$ is SmRw is actually $T_i$'s single thread hit ratio, $Ratio_{hit,single}$.

Case 2 can be broken down into several sub-cases depending on whether a co-running thread accesses the same channel of $R_{i,k}$. We represent each sub-case as a vector. For example, the $l$'th sub-case is $E_{l,\Delta} = \{e_{l,1}, ..., e_{l,j}, ..., e_{l,n}\}$. An element $e_{l,j}$ represents whether thread $T_j$ accesses $R_{i,k}$'s channel: $e_{l,j} = 1$ means yes, and $e_{l,j} = 0$ means no. Clearly, there are $2^n$ sub-cases ($1 \leq l \leq 2^n$). In sub-case $E_{l,\Delta}$, the total number of requests from co-running threads that access $R_{i,k}$'s channel is

$$m_{l,\Delta} = \sum\nolimits_j ReqCnt_{\Delta,j} \times e_{l,j}. \tag{4.12}$$

The probability of sub-case $E_{l,\Delta}$ is then

$$\begin{aligned} P_{l,\Delta} = & Ratio_{hit,single} \times \prod\nolimits_j (e_{l,j} \times (P_{SmBk} + \\ & P_{SmCh}) + (1 - e_{l,j}) \times P_{DfCh}). \end{aligned} \tag{4.13}$$

If there is no SmBk request, then $E_{l,\Delta}$ can be a hit if the row buffer is not automatically closed. If there are SmBk requests, $E_{l,\Delta}$ can still be a hit if the row buffer is not auto-closed and the MC reorders the requests. In short, $E_{l,\Delta}$ is a hit if

the row buffer is not auto-closed:

$$Ratio_{hit,l,\Delta}(case2) = \begin{cases} P_{l,\Delta}, & \text{if } m_{l,\Delta} < D_{ac}, \\ 0, & \text{otherwise.} \end{cases} \tag{4.14}$$

The hit ratio is then the sum of all sub-cases:

$$\begin{aligned} &Ratio_{hit,\Delta} = Ratio_{hit,\Delta}(caseA)+ \\ &Ratio_{hit,\Delta}(caseB) + \sum\nolimits_{l} Ratio_{hit,l,\Delta}(case2). \end{aligned} \tag{4.15}$$

Note that different requests have different BRDs. In other words, for an arbitrary request $R_{i,k}$, it may have different BRDs with different probabilities. Assume the input parameter, the probability of BRD $\Delta$ is $P_\Delta$. Then the hit ratio of thread $T_i$ is the sum of the hit ratios of all its BRDs:

$$Ratio_{hit} = \sum_{\Delta} P_\Delta \times Ratio_{hit,\Delta}. \tag{4.16}$$

**Predicting Miss/Conflict Ratios**

The miss and conflict ratios can be predicted similarly.

### 4.4.3 Predicting Request Latencies

With HMC ratios determined, this section discusses the prediction of HMC latencies. From Figure 4.4, we gain a third key insight into DRAM concurrency and HMC latencies.

*Insight 3: When there is large number of DRAM requests served concurrently, the average latencies of hit/miss/conflict requests are approximately their maximum latencies minus the time that they overlap with other requests' data transfers.*

Assume the maximum latencies of hit/miss/conflict are $Max_{ty}$, $ty \in \{hit, miss, conflict\}$, Insight 3 is essentially

$$Lat_{ty} = Max_{ty} - overlapped\_data\_transfers. \tag{4.17}$$

The maximum latencies are listed in Section 4.2.2. Therefore, we only have to determine the $overlapped\_data\_transers$.

### Hit Latency (Read)

For a hit, the column access time (tCAS in Section 4.2.2) can overlap with any request's data transfer. However, its own data transfer ($tBurst$) requires exclusive access to the data bus. Consequently, its average latency is

$$\begin{aligned} Lat_{r,hit} &= Max_{r,hit} - overlapped\_data\_transfers \\ &= (tCAS + tBurst) - tCAS = tBurst. \end{aligned} \tag{4.18}$$

### Miss Latency (Read)

Because a miss opens a new bank, and because of the FAW limit and adaptive page policy, its overlapped data transfer time varies with the type of overlapped requests. That is, we can rewrite equation (4.17) as

$$Lat_{r,miss} = Max_{r,miss} - \sum_{ty} overlap\_trans\_time_{ty}. \tag{4.19}$$

Because the overlapped data transfer time is the number of overlapped requests multiplied by the data transportation time ($tBurst$), we can further rewrite the equation to

$$Lat_{r,miss} = Max_{r,miss} - \sum_{ty} overlap\_req_{ty} \times tBurst. \tag{4.20}$$

Here, the $overlap\_req_{ty}$ represents the number of type $ty$ requests that overlap with one miss request.

Now the problem of determining miss latency is reduced to the problem of determining the number of overlapped requests of each type. First, consider the case of hits overlapping with a miss. Within a sequence of DRAM requests, $Ratio_{hit}$ of them are hits and $Ratio_{miss}$ are misses. Therefore, for one miss, there are at most $\frac{Ratio_{hit}}{Ratio_{miss}}$ hits. Additionally, FAW limits the maximum number of banks ($MaxBk$) that can be simultaneously accessed. Moreover, because concurrent hits are most likely from different threads and do not access the same bank, the total number of concurrent hits is also limited by $MaxBk$. Assume that the input parameter $rk\_cnt$ is the number of ranks being accessed. Combining all these arguments, we have

$$\begin{aligned} MaxBk &= rk\_cnt \times 4, \\ overlap\_req_{hit} &= min(MaxBk - 1, \frac{Ratio_{hit}}{Ratio_{miss}}). \end{aligned} \tag{4.21}$$

Unfortunately, the number of misses that overlap with another miss cannot be determined using the same approach because there is only one $Ratio_{miss}$. Here, we consider a sequence of $n$ requests from $n$ threads. In this sequence, there are $n \times Ratio_{miss}$ misses. That is, one miss may overlap with $n \times Ratio_{miss} - 1$ misses. Additionally, FAW limit and $MaxBk$ also apply to concurrent miss.

Furthermore, conflicts also require opening new rows, whereas the FAW and adaptive page policy also apply. Therefore, we compute the number of misses and conflicts that overlap with a miss together,

$$\begin{aligned} &overlap\_req_{miss+conf} = min(MaxBk - 1, \\ &n \times (Ratio_{miss} + Ratio_{conf}) - 1). \end{aligned} \tag{4.22}$$

Combining equations (4.20) through (4.22), we have

$$\begin{aligned} &Lat_{r,miss} = (tRCD + tCAS + tBurst) - \\ &(min(MaxBk - 1, \frac{Ratio_{hit}}{Ratio_{miss}}) + min(MaxBk - 1, \\ &n \times (Ratio_{miss} + Ratio_{conf}) - 1)) \times tBurst. \end{aligned} \tag{4.23}$$

**Conflict Latency (Read)**

We predict the average latency of conflicts using the same approach as the miss latency:

$$\begin{aligned} &Lat_{r,conf} = (tRP + tRCD + tCAS + tBurst) - \\ &(min(MaxBk - 1, \frac{Ratio_{hit}}{Ratio_{conf}}) + min(MaxBk - 1, \\ &n \times (Ratio_{miss} + Ratio_{conf}) - 1)) \times tBurst. \end{aligned} \tag{4.24}$$

**Write latencies**

Write HMC latencies can be predicted similarly using the above equations with two changes. First, one extra DRAM cycle besides $tBurst$ is required for data transfer [72]. Second, write recovery time (tWR) should be used as column access time instead of tCAS.

### 4.4.4 Write-to-Read Switching Overhead

When switching from write-to-read, the data bus has to be stalled for tWTR time. Assume the ratio of requests that require a write-to-read switch is $Ratio_{wtr}$ (parameter obtained in Section 4.5). The write-to-read overhead is

$$O_{wtr} = Ratio_{wtr} \times tWTR. \tag{4.25}$$

### 4.4.5 Rank-to-Rank Switching Overhead

When switching between ranks, the data bus has to be stalled for tRTRS time. Assume the ratio of requests that require a rank-to-rank switch is $Ratio_{rtr}$ (parameter obtained in Section 4.5). The rank-to-rank overhead is

$$O_{rtr} = Ratio_{rtr} \times tRTRs. \tag{4.26}$$

At this point we have predicted all unknown variables in equation (4.5), and DraMon model is fully presented.

## 4.5 Obtaining Parameters

As discussed in Section 4.4, DraMon requires input parameters. This section discusses the collection of these parameters. In general, hardware parameters can be collected from data sheets and PCI configurations registers. Software parameters can be collected from memory traces and PMUs. The values of certain software parameters can be also be roughly estimated based on DRAM configuration.

### 4.5.1 Experimental Platform

To demonstrate parameter collection, we use a machine with an AMD Opteron 6174 Processor. This processor has two dies. Each die has six cores which share one 6MB L3 cache and one MC. Each core has 128KB split L1 Cache and 256KB L2 cache. Each MC is connected to two channels of total 12GB memory which is composed of six Samsung M393B5273CH0YH9 DDR3-1333 memory modules. Because this research focuses on predicting the bandwidth of one MC, here we use one-die/six-core of this processor. The machine is running Linux 2.6.32.

| Parameters | Description | Value |
|---|---|---|
| $Size_{mem}$ | size of each DRAM request | 64 Bytes |
| tRCD | row activation time | 13.5 ns |
| tCAS | column access time | 13.5 ns |
| tRP | precharge time | 13.5 ns |
| tBurst | data transfer time | 6 ns |
| tWR | write recovery time | 15 ns |
| tRTRS | rank switching time | 4.5 ns |
| tWTR | write-to-read switching time | 7.5 ns |
| chnl_cnt | number of channels | 2 |
| bk_cnt | number of banks per rank | 8 |
| $D_{ac}$ | row buffer auto-close distance | 4 requests |

Table 4.1: Hardware Parameters

### 4.5.2 Hardware Parameters

Hardware parameters can be collected from data sheets and PCI configurations registers [6, 128]. Table 4.1 gives the description of the hardware parameters, as well as their values of the AMD Opteron 6174 processor.

### 4.5.3 Software Parameters

Table 4.2 gives a list of software parameters. Here we describe two approaches to collect their values. One offline approach that uses both memory-traces and PMU readings. The other approach does not require traces. Instead, it uses PMUs, and it can be applied during execution.

#### Offline Approach

To acquired the parameters of $\Delta$, $P_{\Delta}$, $Ratio_{wr}$, $Ratio_{wtr}$ and $Ratio_{rtr}$, we run each program with one thread and generate its memory trace using Pin [91]. This trace contains the virtual addresses of memory requests, which are translated to physical addresses using the page table exported by Linux kernel. Each physical address is later translated to a DRAM address, which includes the channel, rank, bank, row and column addresses [6]. This translated trace is input into an in-house cache simulator to generate DRAM requests.

We analyze the DRAM request trace to collect bank reuse distances and their probabilities ($\Delta$ and $P_{\Delta}$), as well as $Ratio_{wr}$, $Ratio_{wtr}$, and $Ratio_{rtr}$. To acquire

| Parameters | Description |
|---|---|
| $Rate_{issue,single}$ | single thread issue rate |
| $\Delta$ and $P_{\Delta}$ | Bank reuse distances and their probs |
| $Ratio_{hit,single}$ | single thread hit ratio |
| $Ratio_{miss,single}$ | single thread miss ratio |
| $Ratio_{conf,single}$ | single thread conflict ratio |
| $P_{SmRw}$ | same-row accessing probability |
| $P_{SmBk}$ | same-bank-diff-row accessing prob. |
| $P_{SmCh}$ | diff-bank-same-channel accessing prob. |
| $P_{DfCh}$ | different channel accessing probability |
| $Ratio_{wr}$ | Write request ratio |
| $Ratio_{wtr}$ | write-to-read switching request ratio |
| $Ratio_{rtr}$ | rank switching request ratio |
| $rk\_cnt$ | number of ranks accessed |

Table 4.2: Software Parameters

| Parameters | PMU |
|---|---|
| $Rate_{issue,single}$ | DRAM_ACCESSES_PAGE, HW_CPU_CYCLES |
| $Ratio_{hit,single}$ | DRAM_ACCESSES_PAGE:HIT |
| $Ratio_{miss,single}$ | DRAM_ACCESSES_PAGE:MISS |
| $Ratio_{conf,single}$ | DRAM_ACCESSES_PAGE:CONFLICT |
| $Ratio_{wr}$ | MEM_CONTROLLER_REQ:WRITE_REQ |
| $Ratio_{wtr}$ | MEM_CONTROLLER_TURN:WRITE_TO_READ |
| $Ratio_{rtr}$ | MEM_CONTROLLER_TURN:CHIP_SELECT |

Table 4.3: Collecting program parameters from PMUs

single thread HMC ratios, we input the trace into an in-house DRAM simulator.

To obtain $P_{SmRw}$, $P_{SmBk}$, $P_{SmCh}$ and $P_{DfCh}$, we run the program with two threads and collect their memory traces using the above approach. Then we run both traces with our DRAM simulator to generate these probabilities. Currently, we collect traces with up to 75 million requests. The recording and processing of a trace takes about 30 minutes. It may be possible to use shorter traces for online processing as suggested by previous research [160].

For the last missing parameter, $Rate_{issue,single}$ , it can only be acquired using the PMU. On the AMD processor we use, this PMU counter is DRAM_ACCESSES_PAGE.

| Benchmarks | $P_{SmRw}$ | $P_{SmBk}$ | $P_{DfCh}$ | BRDs |
|---|---|---|---|---|
| streamcluster. | 0.01% | 6.26% | 49.67% | 1(78%), 8(22%) |
| facesim | 0.00% | 5.57% | 50.20% | 1(90%), 2(10%) |
| canneal | 0.01% | 6.20% | 50.67% | 1(93%), 8( 7%) |
| fluidanimate. | 0.01% | 6.28% | 48.93% | 1(78%), 4(22%) |

Table 4.4: $P_{SmRw}$, $P_{SmBk}$, $P_{Dfch}$ and BRDs of four PARSEC benchmarks.

**Run-time Approach**

Most software parameters can also be collected from PMUs. Table 4.3 gives a list of software parameters and their corresponding PMUs on the AMD processor we used.

Unfortunately, there is no PMU that provides values for $P_{SmRw}$, $P_{SmBk}$, $P_{SmCh}$, $P_{DfCh}$ and bank reuse distances. However, from memory traces, we discovered that most programs share common values for these variables. Table 4.4 gives the values of these parameters of four PARSEC benchmarks (for BRD values, a $x(y)$ represents a BRD of $x$ with probability $y$). The table shows that all benchmarks have $P_{SmBk}$ of around 6%. The reason for this similarity is the memory interleaving behavior of the MC. Currently, when allocating memory pages, MC distributes the pages evenly among the banks for better performance. For example, when there is only one conceptual rank of memory used, there are 16 banks involved (8 per channel). Therefore, the probability that one bank is accessed by a thread is $\frac{1}{16}$. And the probability that two threads accessing the same bank is $\frac{1}{16} \cdot \frac{1}{16} \cdot 16 = 6.25\%$. Because of this memory interleaving behavior, we use $\frac{1}{bk\_cnt \cdot rk\_cnt \cdot chnl\_cnt}$ as $P_{SmBk}$, where $rk\_cnt$ can be acquired from the OS. Similarly, we use 0% for $P_{SmRw}$ because two threads rarely access the same row, and 50% for $P_{DfCh}$ because there are two channels. $P_{SmCh}$ is $1 - P_{SmRW} - P_{SmBk} - P_{DfCh}$.

For BRDs, most programs have a BRD of one with a high probability. The reason for this similarity is data locality, i.e., consecutive requests are likely to access the same row. Consequently, we assume a sequential access pattern for DraMon-R. On our machine, the channel-interleaving dictates that every eight consecutive same-row requests start accessing a new channel. That is, among the eight requests, the first request has a BRD of eight with probability $\frac{1}{8} = 12.5\%$; the reset seven requests have a BRD of one with probability $\frac{7}{8} = 87.5\%$. That is, we use 1(87.5%) and 8(12.5%) as BRDs for run-time prediction.

## 4.6 Experimental Evaluation

We implemented both DraMon-R and DraMon-R models, and evaluated their accuracy and fidelity on a real multi-core machine. This section presents the evaluation results.

### 4.6.1 Experimental Setup

Our goal is to evaluate the accuracy and fidelity of DraMon. Here we use the same platform specified in Section 4.5.1. Our experiments use 10 benchmarks from PARSEC2.1 and all 10 benchmarks from NPB-OMP3.3.1 [15, 75]. The results of three PARSEC benchmarks *bodytrack*, *dedup* and *x264* are not reported as they are I/O bound and have very limited bandwidth requirement.

Two kernel benchmarks are also considered for their high bandwidth requirements and wide uses: *fft*, a Fast Fourier Transform program [149], and *bw_mem* from lmbench3 benchmark suite [98]. For PARSEC and NPB benchmarks, the largest executable input sets, "native" and "D", are used. All benchmarks are compiled using GCC/GFortran 4.4.3. PARSEC and kernel benchmarks are compiled with O3 optimization flag, and NPB benchmarks are compiled with O1 flag. These benchmarks cover a variety of memory access patterns, including read/write requests, single-bank/multi-bank accesses and streaming/random accesses.

For each benchmark, we predict its HMC ratios and bandwidth usages when it runs with two to six cores/threads using one MC. Then we compare the predicted values with the real values obtained from PMUs (Table 4.2), and report the accuracy of our predictions. Additionally, we compare DraMon to a state-of-the-art, multiple linear and logarithmic regressions based bandwidth model [81].

We define the bandwidth prediction accuracy as

$$Accuracy_{bw} = 100\% - \left| \frac{BW_{real} - BW_{predicted}}{BW_{real}} \right|. \tag{4.27}$$

For HMC ratios predictions, we leverage the *multinomial likelihood* $L$ from the likelihood theory [134],

$$\begin{aligned} D_{KL} &= \sum_{ty} Ratio_{ty,real} \times \log_2 \left( \frac{Ratio_{ty,real}}{Ratio_{ty,predicted}} \right), \\ L &= 2^{-D_{KL}}, \quad ty \in \{hit, miss, conflict\}. \end{aligned} \tag{4.28}$$

| Benchmarks | Memory behavior | Accuracy $L$ | |
|---|---|---|---|
| | | DraMon-T | DraMon-R |
| streamcluster | read, single-rank, streaming | 99.83% | 99.80% |
| facesim | read, single-rank, streaming | 99.34% | 98.26% |
| canneal | read/write, single-rank | 98.65% | 98.51% |
| lu.D | read/write, multi-rank | 98.50% | 97.57% |
| mg.D | read/write, multi-rank | 99.64% | 98.64% |
| sp.D | read/write, multi-rank | 99.07% | 97.90% |
| fft | read/write, single-rank, streaming | 99.83% | 99.80% |
| bw_mem | read, single-rank, streaming | 99.83% | 99.80% |
| Average | | 99.17% | 98.55% |

Table 4.5: DRAM contention (HMC ratios) prediction accuracy.

Intuitively, $L$ represents the probability that a model is accurate if the model predicts a probability distribution. Here, we simply refer to $L$ as "HMC ratios prediction accuracy."

### 4.6.2 DRAM Contention (HMC Ratios) Prediction

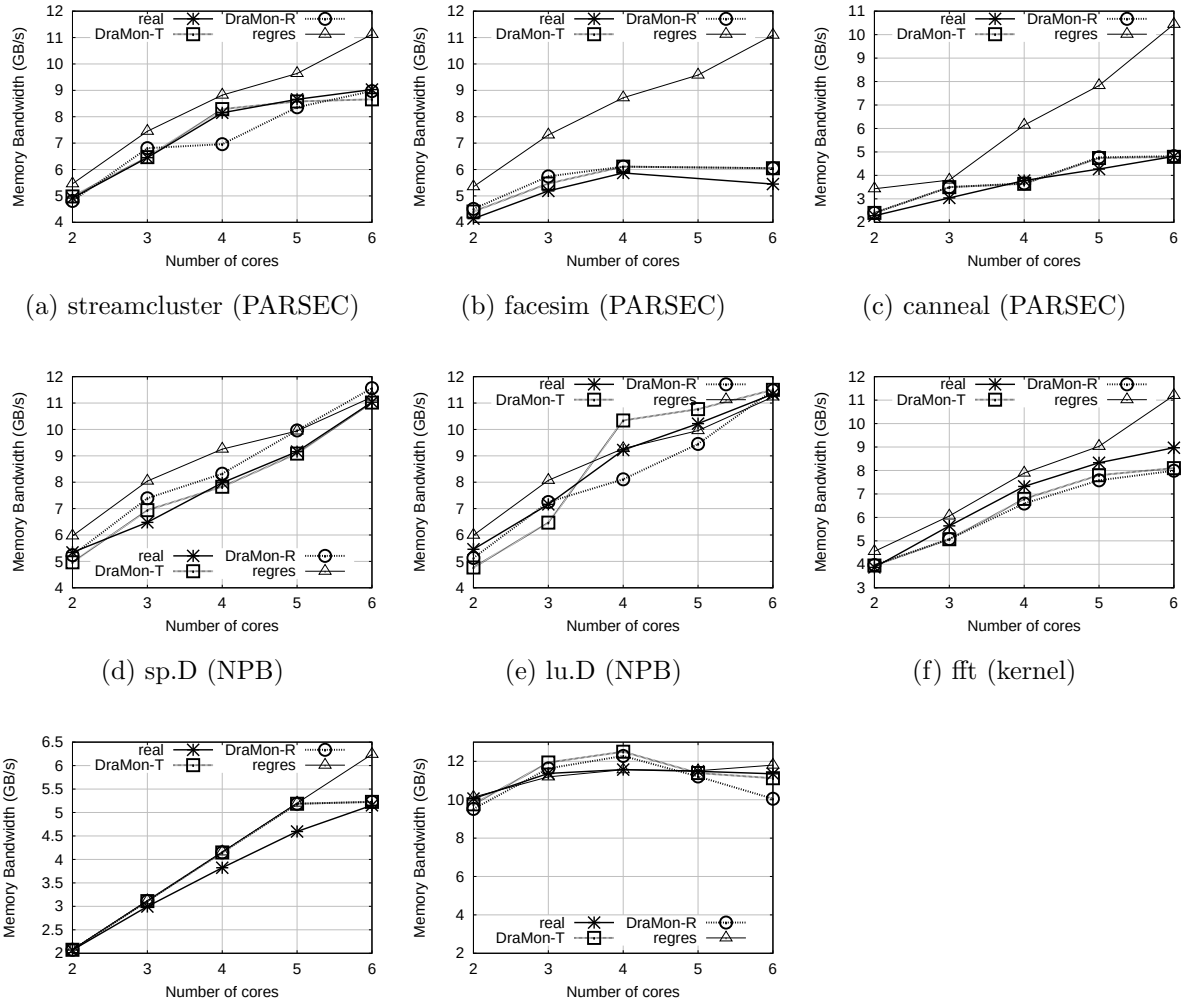
Table 4.5 gives the accuracy of DRAM contention prediction for the eight most memory-intensive benchmarks. As the table shows, DraMon is very accurate for memory-intensive programs with a wide range of memory behaviors.

For the rest benchmarks (other than the eight in Table 4.5), the average accuracy of DraMon-T is 96.99%, and the average accuracy of DraMon-R mode is 96.85%. The highest accuracy of DraMon-T is 99.53% (*fluidanimate*). The highest accuracy of DraMon-R is 99.48% (*raytrace*). The lowest accuracy of both models is 92.46%(*blackscholes*). The overall accuracies of DraMon-T and DraMon-R for all benchmarks are 97.95% and 97.61%, respectively.

### 4.6.3 Bandwidth Usage Prediction

#### Bandwidth Results

Figure 4.7 composes the bandwidth prediction results for the eight memory-intensive benchmarks. The result of *facesim* for five cores/threads is missing, because PARSEC benchmark suite does not provide execution configuration for *facesim* to execute with 5 cores/threads.




(a) streamcluster (PARSEC) (b) facesim (PARSEC) (c) canneal (PARSEC)

(d) sp.D (NPB) (e) lu.D (NPB) (f) fft (kernel)

(g) mg.D (NPB) (h) bw_mem (lmbench)

Figure 4.7: Bandwidth prediction results.

DraMon-T has an average accuracy of 94.7%, and DraMon-R has an average accuracy of 93.37%. These results demonstrate that DraMon can accurately predict bandwidth usage for programs with a wide range of bandwidth requirements and memory behaviors. The highest accuracies of DraMon-T and DraMon-R are 98.31% (*streamcluster*) and 95.07% (*mg.D*), respectively. The lowest accuracies of DraMon-T and DraMon-R are 91.49% (*lu.D*) and 90.30% (*fft*), respectively.

The bandwidth usages of the rest benchmarks increase linearly with the number of cores/threads. Both models have a high average accuracy of 97.61%, with the highest accuracy of 99.31% (*blackscholes*) and the lowest accuracy of 90.43% (*bt.D*). The overall accuracies of DraMon-T and DraMon-R for all benchmarks are 96.32% and 95.73%..

**Comparing to Regression Model**

In Figure 4.7, results labeled with "regres" are the predictions from a regression-based model [81]. The average accuracy of this model is 77.61%, which is significantly lower than DraMon. Its worst case accuracy is 44.37% (*facesim*), which is also lower than DraMon's lowest accuracy (90.30%). Except for *lu.d* and *bw_mem*, six benchmarks have higher accuracies using DraMon. For *bw_mem*, the regression model has a higher accuracy because it is trained using a micro-benchmark that has a similar behavior as *bw_mem*.

### 4.6.4 Execution Time of the Run-time Model

The run-time model requires reading seven PMUs (Table 4.3). We collect the PMU readings for 0.5 seconds of execution. We implemented DraMon using C. The average time for computing the bandwidth of one core/thread configuration is 0.03 seconds. For each benchmark, five configurations are predicted. Including parameter collection, the total prediction time is 0.65 seconds, which only adds 0.5% to the execution time of our benchmarks in average.

## 4.7 Discussion

This section discusses issues that affect DraMon's accuracy.

### 4.7.1 Prefetcher Impact

In our experiments, the memory prefetcher is enabled. This prefetcher fetches a stream of data from memory if a stride memory access pattern is detected [6]. We observe that this prefetcher has a high prefetching accuracy. It also adaptively decreases the number of prefetching requests in case of heavy DRAM contention [6]. Therefore, the existence of this prefetcher does not affect DraMon's accuracy. However, a less accurate or non-adaptive prefetcher may need to be modeled separately.

### 4.7.2 DRAM Refresh Impact

DRAM cells need periodical refreshing to retain their data, which can degrade DRAM performance. The DRAM module used in our experiments requires that each bank spend $160ns$ on refreshing every $7.8us$ [128]. Therefore, the DRAM refresh has a theoretical overhead of $\frac{160ns}{7.8us} = 2\%$ [152]. This overhead may be lower than 2% if rank-level parallelism happens [72]. This low overhead does not significantly impact the accuracy of DraMon. Additionally, DRAM refresh overhead can be mitigated for high density DRAM modules [109, 112, 142].

### 4.7.3 Cache Impact

Because this research focuses on DRAM, predicting cache performance is beyond its scope. However, because DraMon is evaluated on a real machine, cache does have some impact on our results. Fortunately, memory-intensive benchmarks already have high cache miss rates, and their memory behaviors are not affected by cache contention.[2]

However, four benchmarks, *ferret*, *swaptions*, *freqmine*, and *ep.D* which have very low bandwidth requirements, are affected by cache contention or data sharing. Predicting their bandwidth usages requires a cache model [145, 159]. Because these benchmarks' bandwidth usages depend on cache, their results are not included in the average accuracies in Section 4.6.

### 4.7.4 Generalization

Using DraMon on a new platform requires updating its input parameters accordingly. The hardware parameters can be updated based on the new hardware configuration.

[2]These benchmarks are known as *devils* by previous research [161].

For DraMon-T, the software parameters can still be obtained from memory traces. For DraMon-R, the corresponding PMUs should be identified on the new platform. Software parameters which cannot be obtained from PMUs (i.e., SmRw, SmBk, SmCh and DfCh probabilities, and bank reuse distances) are determined by the DRAM configuration (banks/ranks/channels), as well as the channel-interleaving scheme as discussed in Section 4.5.3.

## 4.8 Summary

This chapter presented DraMon, a model that predicts the bandwidth usage of multi-threaded programs on real machines with high accuracy. DraMon can be directly employed to predict the optimal core allocation of local memory bandwidth limited multi-threaded applications. It also can be used to improve DRAM system design and memory allocation algorithms, as well as addressing other problems involving DRAM contention.

We demonstrated that accurately predicting memory bandwidth requires predicting DRAM contention and DRAM concurrency, which both can be predicted with high accuracy and in short computation time using probability theory. We also identified the hardware and software factors that should be considered for bandwidth prediction. These parameters can be collected from memory trace, as well as PMUs for run-time prediction. When evaluated on a real machine, DraMon shows high average accuracies of 98.55% and 93.37% for DRAM contention and bandwidth predictions, with only 0.50% overhead on average.

# Chapter 5

# Predicting Inter-node Memory Bandwidth Usages and Optimal Core Allocations

## 5.1 Introduction

The previous chapter (Chapter 4) presented a model that predicts the optimal core allocation for local memory bandwidth limited applications. However, there are also applications that are limited by inter-node memory bandwidth as shown in Chapter 3. For these applications, an optimal core allocation model that considers the inter-node memory bandwidth is required. Designing such a model with high accuracy is very challenging, because there are a large variety of bandwidth-impacting factors to be considered. More specifically, the following challenges must be addressed:

1. The first challenge is to properly handle various bandwidth limiting factors for high accuracy bandwidth usage prediction. The large variety of these interacting factors, such as local-remote memory access contention and bit-width of a link, makes prediction very difficult. The impact of these factors also varies with the application and platform, further complicating this challenge.

2. The second challenge is to properly handle the heterogeneity and non-uniformity within a program and a NUMA system for high accuracy bandwidth usage prediction. As shown by Figure 1.2 on page 3, and by the examples in Section 3.2, the heterogeneity and non-uniformity are one of the major determining factors for bandwidth usages and optimal core allocations. They require that each

inter-node connection and each node to be treated differently, which greatly complicates the prediction problem.

3. The third challenge is to predict the optimal core allocation in reasonable time. Because of the heterogeneity and non-uniformity, core allocation on NUMA machines is not only about the numbers of cores, but also about the location (which node) of the cores. There can be millions of different core allocations to consider. For instance, an existing NUMA machine with eight six-core nodes has about $7^8 = 5,764,801$ possible core allocations.

In this chapter, we present two models, NuMem and NuCore, for memory-intensive multi-threaded applications. NuMem predicts the inter-node and total memory bandwidth usage of an application on a large-scale NUMA machine. It achieves high accuracy and low overhead by reducing the prediction problem to a Mixed Integer Programming (MIP) problem. It expresses the bandwidth limiting factors as linear constraints. These factors include an application's bandwidth requirements, the contention between local and remote accesses, the contention for shared inter-node links, and the physical limit (frequency and bit-width) of the inter-node links. Each link is evaluated with its own constraints to reflect the heterogeneity and non-uniformity. NuMem also expresses the memory bandwidth usage as an objective function of the MIP problem, so that it can be predicted by solving the MIP problem with a MIP solver.

Based on NuMem, we developed NuCore to consider additional non-linear constraints to predict the optimal core allocation with MIP. We present a technique that can convert these non-linear constraints into linear constraints. NuCore also combines the goals of minimizing core allocations and maximizing bandwidth usage into one objective function. The optimal core allocation is then predicted using a MIP solver without the need to examine all millions of core allocations.

We evaluated our models on two large-scale NUMA platforms presented in Section 3.1. Experimental results show that NuMem is highly accurate with an average bandwidth prediction error of less than 10% for memory-intensive benchmarks. NuCore correctly predicted the optimal core allocations of 19 out of 22 benchmarks. For the other three benchmarks, NuCore's predictions differed by at most one core per node from the experimentally determined optimal core allocations. The predicted optimal core allocations perform within 1.0% of the real optimal on average. The predicted core allocations provides up to 3.34 times speedup over the use-all-cores

allocation, while using only 12.5% of all available cores. The average speedup of predicted core allocations is 1.27 for memory-intensive benchmarks. More importantly, this speedup is achieved using much fewer resources, i.e., the predicted core allocations use only 75.6% of all available cores on average. The results also show that NuCore can be solved by a MIP solver in 0.02 seconds, suggesting that NuCore can be applied at run-time.

The contributions of this chapter include:

1. A model, NuMem, that predicts the memory bandwidth usage of one core allocation with high accuracy using MIP.

2. A model, NuCore, that predicts the optimal core allocation with high accuracy and low overhead using MIP.

3. To the best of our knowledge, NuMem and NuCore are the first analytical model to mathematically express the NUMA bandwidth limiting factors to predict bandwidth usage and optimal core allocation. The technique of expressing non-linear bandwidth limiting factors and contentions as linear constraints, and expressing prediction or optimization goals as objective functions may also apply to other memory-related NUMA optimization problems.

4. A thorough experimental evaluation of the accuracy, overhead, and performance benefits of our models, using two large scale NUMA machines with 22 benchmarks.

This chapter is organized as follows. Section 5.2 provides an overview of our models. Section 5.3 presents NuMem in detail. Section 5.4 describes NuCore in detail. Section 5.5 gives the experimental evaluation of our models. Section 5.6 discusses related issues, and Section 5.7 summarizes the work.

## 5.2 Overview of NuMem and NuCore

This section gives an overview of NuMem and NuCore. NuMem predicts bandwidth usages with a MIP formulation which consists of a set of constraint functions and an objective function. Based on NuMem, NuCore predicts optimal core allocations by considering additional constraint functions and objective functions.

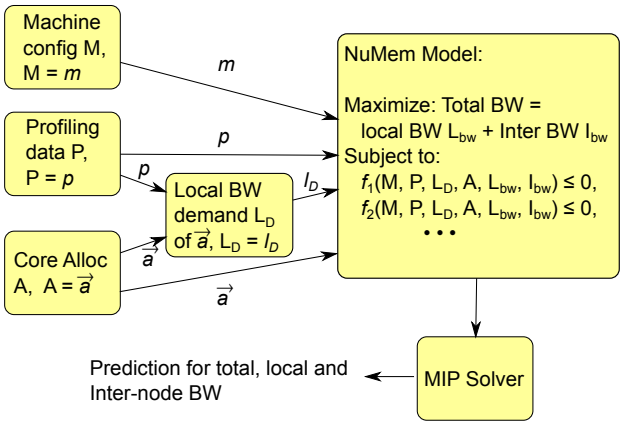



Figure 5.1: Overview of the NuMem model.

### 5.2.1 NuMem Overview

The bandwidth usage of an application is limited by various factors. Under these constraints, the application uses as much bandwidth as possible. Similarly, in a MIP problem, which is a linear programming problem with integer variables, the maximum (optimal) value of a linear objective function is determined under a set of linear constraints. This similarity makes it possible for NuMem to use MIP to predict bandwidth usage. The challenge here is to express the bandwidth constraints and bandwidth usage as linear functions.

Figure 5.1 gives the overview of NuMem. NuMem predicts the memory bandwidth usage for a specific multi-threaded application executing on a specific NUMA machine with a specific core allocation. The input parameters to NuMem are:

1. The configuration of the NUMA machine, denoted by $M$, such as the node connection topology and the maximum bandwidth of the inter-node links.

2. Profiling data, denoted by $P$, that describes the memory access behavior of the application. To collect profiling data, at the beginning of each program phase, the application is allocated with one core per node on all nodes and executed briefly, while hardware performance monitoring units (PMU) are used to collect its bandwidth usage. This profiling can be performed online during application execution. For this work, we found a phase detection technique similar to prior work is sufficient [60]. More sophisticated phase-detection techniques can also be used [129, 133].

3. The core allocation, denoted by $A$, used by the application. Recall that $A$ is

essentially a vector $\{a_0, a_1, \ldots, a_i, \ldots, a_N\}$, where $a_i$ represents the number of cores allocated on $node_i$.

4. The prediction of the local memory bandwidth demand, denoted by $L_D$, of the application under core allocation $A$. A local bandwidth model, DraMon, is used to obtain this prediction [154]. The local memory bandwidth usage is then predicted based on the DraMon's prediction with consideration of the contention from remote memory accesses.

The values of $M$, $P$, $A$ and $L_D$, denoted by $m$, $p$, $\vec{a}$ and $l_d$, describes the characteristics of a particular application, machine and core allocation. These values are passed into NuMem, which maximizes the total bandwidth subject to a set of bandwidth constraints (denoted by $f_i$ in Figure 5.1), to predict the memory bandwidth usages using a MIP solver.

### 5.2.2 NuCore Overview

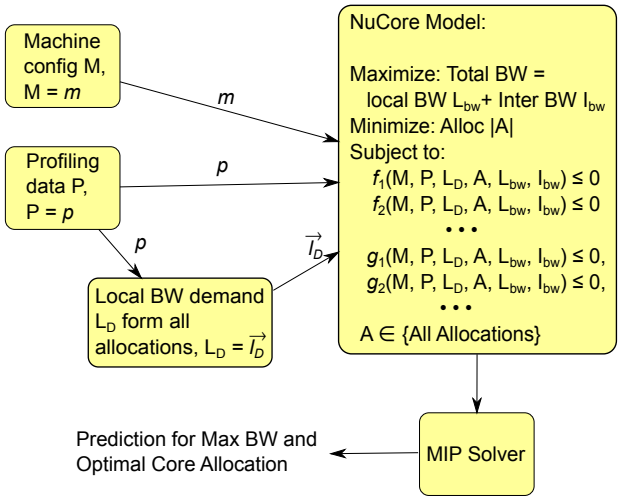



Figure 5.2: Overview of the NuCore model.

The naive method of predicting the optimal core allocation is to use NuMem to predict the bandwidth usages of all possible core allocations to determine the minimum core allocation with the maximum bandwidth usage. This approach, however, has high overhead. Here, based on NuMem, we designed NuCore to consider all possible core allocations by including additional constraints and optimization objectives. Figure 5.2 gives an overview of NuCore, where the additional constraints are denoted by $g_i$. The new constraints and objective are:

1. The core allocation $A$ is no longer an input, but a value to predict. Its value can be any of the possible core allocations.

2. The prediction of the local memory bandwidth demand, $L_D$, should consider the predictions for all possible core allocations. However, these new predictions introduce non-linear constraints, which can be converted into linear constraints with a technique described in Section 5.4.

3. The optimization goals are to maximize bandwidth usage and minimize the core allocation.

The actual values of $M$, $P$, and $L_D$ (i.e., $m$, $p$, and $\vec{l_d}$) are passed into NuCore to predict the optimal core allocation with a MIP solver for a specific application and machine. After an optimal core allocation prediction is made, the application can be reconfigured to use the predicted core allocation with dynamic thread-changing techniques [1, 68, 86, 104].

## 5.3 Predicting Bandwidth Usage

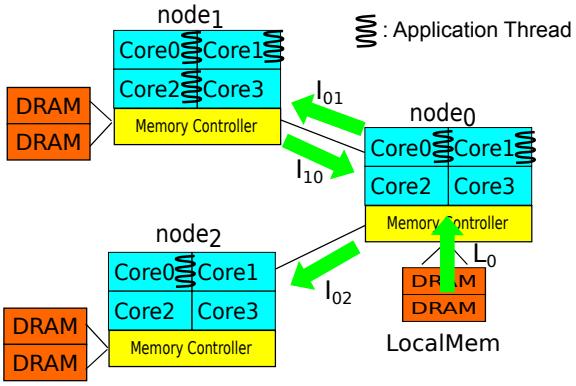



Figure 5.3: A typical case with both local memory accesses and inter-node memory accesses from different nodes.

A NUMA system typically has both local memory accesses and remote memory accesses. The remote memory accesses can also use any link in the system. Figure 5.3 depicts a case where a multi-threaded application is running on a three-node NUMA system. The application is using two cores on $node_0$, three cores on $node_1$ and one core on $node_2$. That is, its core allocation is $\{2, 3, 1\}$. The threads on $node_0$ are accessing $node_0$'s local memory with a bandwidth usage $L_0$. They are also accessing

$node_1$'s memory with a bandwidth usage of $I_{10}$. The threads on $node_1$ and $node_2$ are accessing the memory of $node_0$ with a bandwidth usage $I_{01}$ and $I_{02}$. Here, predicting the bandwidth usage involves predicting the values of $L_0$, $I_{10}$, $I_{01}$ and $I_{02}$, under several bandwidth constraints.

Consider the general case where a multi-threaded application is executing on a NUMA machine:

- Let $N$ denote the number of nodes on the machine.
- Let $A = \{a_0, \ldots, a_N\}$ denote the core allocation of the application, where $a_i$ is the number of cores allocated on $node_i$.
- Let $L_i$ denote the local bandwidth usage on $node_i$.
- Let $I_{ji}$ denote the inter-node bandwidth usage of the application when data is sent from $node_j$'s memory to $node_i$.

Predicting the bandwidth usage of the application running with core allocation $A$ is essentially predicting the values of $L_i$ for each node $node_i$, and the values of $I_{ji}$ for each node pair $node_i$ and $node_j$. Previous research efforts showed that these bandwidth usages have the following constraints [20, 94]:

1. The maximum rates that an application consumes and generates data when no resource contention is considered.
2. The physical bandwidth limit of an inter-node link determined by its frequency and bit-width.
3. The contention between the remote accesses that share a link. For example, the $I_{01}$ and $I_{10}$ in Figure 5.3.
4. The contention between the local and remote memory requests accessing the memory of the same node, e.g., in Figure 5.3, $L_0$, $I_{01}$ and $I_{02}$ contend for $node_0$'s memory.

We express these factors and the bandwidth usages as linear constraint functions and objective functions. The following sections discuss the impact of these factors individually, and derive equations that express these factors and the bandwidth usages as linear constraints and objective functions.

### 5.3.1 Constraint 1: Maximum Data Rate

The bandwidth usage of an application depends on the maximum rates that it can issue memory requests. The inter-node bandwidth usage from $node_j$ to $node_i$, $I_{ji}$, includes the usages of the responses to the read requests from $node_i$, and the write requests from $node_j$. Let $I_{r,ji,solo}$ denote the bandwidth usage of an application running on $node_i$ and read-accessing $node_j$ when it uses one (solo) core on $node_i$. The $I_{r,ji,solo}$ is acquired from profiling. Moreover, assume only one thread is executed on a core at a time. Intuitively, if an application with one core/thread is using $I_{r,ji,solo}$ bandwidth, its bandwidth usage ($I_{r,ji}$) with $a_i$ cores/threads is no larger than $a_i$ times $I_{r,ji,solo}$:

$$I_{r,ji} \leq a_i \times I_{r,ji,solo}. \tag{5.1}$$

Similarly, if the threads on $node_j$ are also writing to $node_i$, then the write-bandwidth usage $I_{w,ji}$ is limited by,

$$I_{w,ji} \leq a_j \times I_{w,ji,solo}. \tag{5.2}$$

The total inter-node bandwidth usage from $node_j$ to $node_i$ is the sum of the read and write bandwidth:

$$I_{ji} = I_{r,ji} + I_{w,ji}. \tag{5.3}$$

Let $L_{D,i}$ denote the local bandwidth demand, which is predicted using the local bandwidth model, DraMon [154]. The local bandwidth usage of $node_i$, denoted by $L_i$, is also limited by its local bandwidth demand:

$$L_i \leq L_{D,i}, \tag{5.4}$$

### 5.3.2 Constraint 2: Physical Limit

The bandwidth usage from $node_j$ to $node_i$, $I_{ji}$, is also limited by the frequency and the number of bits of the inter-node link that connects $node_j$ to $node_i$. Let $UniMax_{ji}$ denote this physical limit. This constraint is

$$I_{ji} \leq UniMax_{ji}, \tag{5.5}$$

where $UniMax_{ji}$ is acquired from machine configuration, and its value can be determined theoretically based on the frequency and bit-width of the link. However, the

practical maximum bandwidth is usually smaller than the theoretical value for two reaons. First, there are overheads from the packet header and CRC code for each data packet [66, 148]. Second, the link is also used to send snooping messages. Therefore, we used the bandwidth measurement benchmark suite *lmBench* to determine the value of $UniMax_{ji}$ experimentally [98, 120].

### 5.3.3 Constraint 3: Contention for Same Link

In a NUMA machine, $node_i$ and $node_j$ can access each other simultaneously over the same inter-node link. These accesses contend for the shared link. That is, the sum of the bandwidth usages of $I_{ji}$ and $I_{ij}$ are limited by this contention. Let $BiMax_{ij}$ denote the maximum bi-directional bandwidth of the physical link between $node_i$ and $node_j$. This constraint is then

$$I_{ji} + I_{ij} \leq BiMax_{ij}. \tag{5.6}$$

$BiMax_{ij}$ is acquired from the machine configuration. Its value is also determined experimentally using the *lmBench*.

### 5.3.4 Constraint 4: Local/Remote Contention

For memory-intensive applications, because both the remote requests and local requests to a node are accessing that node's local memory, their bandwidth is limited by the maximum available bandwidth of that node's memory. Our experimental results reveal that the maximum of the sum of the outgoing inter-node bandwidth of a node has a linear relationship with its local bandwidth demand. Figure 5.4 depicts this linear relationship for an Intel X7550 processor. The $y$-intercept and the slope of the linear equation, $\alpha_i$ and $\beta_i$, are 20.26 and 0.24 for this machine (determined with *lmbench*). The correlation coefficient is 0.96 indicating a strong linear relationship.

Let $N$ denote the number of nodes, and $L_{D,i}$ denote the local bandwidth demand on $node_i$ predicted with the DraMon model [154]. Based on this linear relationship, the constraint of the local and remote access contention can be expressed as

$$\sum_{j=1}^{N} I_{ij} + \beta_i \times L_{D,i} \leq \alpha_i, \tag{5.7}$$

We deduce that this linear relationship is a reflection of the processor's attempt

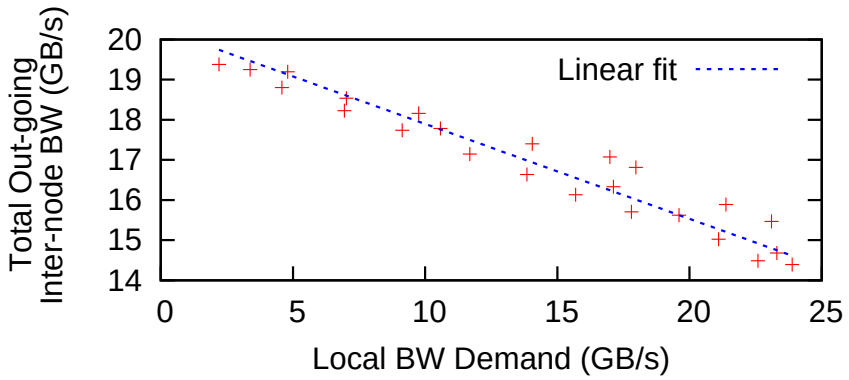



Figure 5.4: Linear relationship of the total outgoing bandwidth and the local bandwidth demand of an Intel processor.

to ensure every requesting source, including the inter-node links and the local cores, gets a fair share of its memory bandwidth. Intuitively, $\alpha_i$ represents the maximum bandwidth of $node_i$'s memory, and $\beta_i$ represents the lowest share of $node_i$'s bandwidth reserved for local cores. This insight leads to another constraint that binds the local bandwidth usage on any $node_i$, shown in Equation (5.8). Intuitively, the sum of the total out-going bandwidth and the local bandwidth of $node_i$, cannot exceed its maximum bandwidth $\alpha_i$.

$$\sum_{j=1}^{N} I_{ij} + L_i \leq \alpha_i. \tag{5.8}$$

### 5.3.5 Handling Multi-hop Links

Two nodes of a NUMA machine can be indirectly connected through other nodes, i.e., these two nodes are indirectly connected through several physical links. The inter-node memory accesses between these two nodes affect the bandwidth usages of the links that connect them. For example, in Figure 5.3, $node_1$ and $node_2$ are indirectly connected through $node_0$.

Without loss of generality, assume two nodes $node_l$ and $node_k$ are connected using a virtual link through $d$ nodes $l, l+1, ..., l+d, k$. The bandwidth usage $I_{lk}$ is also subject to the physical limit and bi-directional contention as described in Equation (5.5) and (5.6). $UniMax_{lk}$ and $BiMax_{lk}$ are also determined experimentally using *lmBench*. Note that $UniMax_{lk}$ and $BiMax_{lk}$ are smaller than any of its physical links because the virtual link is slower than any of its physical links.

Additionally, because the virtual link sends data through its physical links, its

bandwidth usage $I_{lk}$ should also be added to its physical links:

$$I_{ji} = I_{ji,r} + I_{ji,w} + \sum_{l,k} I_{lk}, \forall\ node_l \text{ and } node_k \\ \text{connected through link } j \rightarrow i. \tag{5.9}$$

### 5.3.6 Objective Function

The objective function for maximizing the sum of local and inter-node bandwidth is

$$\text{maximize: } \sum_i L_i + \sum_{i,j} I_{ij}. \tag{5.10}$$

### 5.3.7 NuMem Summary

Equation 5.11 summarizes the linear constraints and objective function of NuMem. A MIP solver can solve this problem with predictions of the values of each $L_i$ and $I_{i,j}$.

$$\begin{aligned} &\text{maximize: } \sum_i L_i + \sum_{i,j} I_{ij} \\ &\text{subject to: } \forall\ i,j\text{:} \\ &\quad \text{constraint}_1\text{: } I_{r,ji} \leq a_i \times I_{r,ji,solo}, \\ &\qquad\qquad I_{w,ji} \leq a_j \times I_{w,ji,solo}, \\ &\qquad\qquad I_{ji} = I_{ji,r} + I_{ji,w} + \sum_{l,k} I_{lk}, \forall\ node_l \text{ and } node_k \\ &\qquad\qquad \text{connected through link } j \rightarrow i., \\ &\quad \text{constraint}_2\text{: } I_{ji} \leq UniMax_{ji}, \\ &\quad \text{constraint}_3\text{: } I_{ji} + I_{ij} \leq BiMax_{ij}, \\ &\quad \text{constraint}_4\text{: } \sum_{j=1}^{M} I_{ij} + \beta_i \times L_{D,i} \leq \alpha_i, \\ &\qquad\qquad \sum_{j=1}^{M} I_{ij} + L_i \leq \alpha_i \end{aligned} \tag{5.11}$$

## 5.4 Predicting Optimal Core Allocation

This section discusses the optimal core allocation prediction model, NuCore. NuMem is only used to predict the inter-node bandwidth for one core allocation. As mentioned

in Section 5.2, to predict the optimal core allocation, other core allocations must be considered. More specifically, the following constraints and objective function are needed:

1. The number of allocated cores of each node can be any number from zero to the maximum available cores. Let $n_i$ denote the maximum number of cores on $node_i$. The core allocation $a_i$ can be any number between 0 and $n_i$, i.e.,

$$0 \leq a_i \leq n_i. \tag{5.12}$$

2. The local bandwidth prediction should be extended to consider predictions to other core allocations.

3. The objective function should reflect the goals of both maximizing total bandwidth and minimizing core allocation.

Although Equation (5.12) alone suffices for the first set of additional constraints, the last two additions require further discussion, which is provided in the following sections.

### 5.4.1 Handling Local Bandwidth Demands

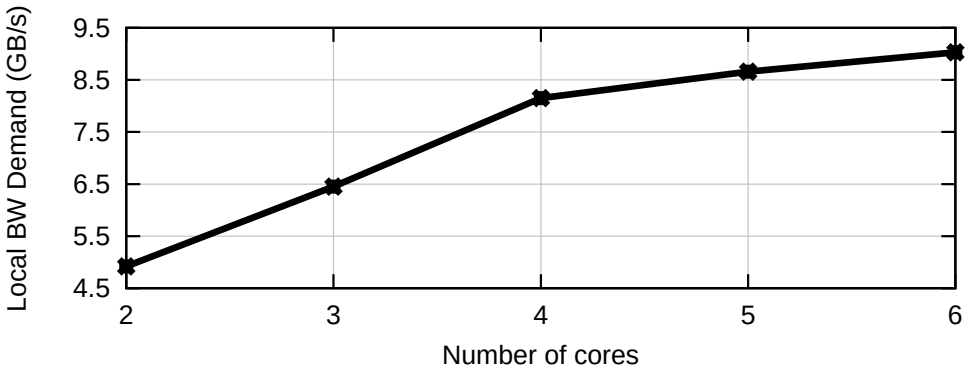



Figure 5.5: The non-linear local bandwidth usage of *streamcluster* on an AMD six-core node.

The local bandwidth demand $L_{D,i}$ of $node_i$ is a function of the numbers of cores used on $node_i$. Unfortunately, this function is not linear due to the contention for DRAM row buffers [154]. Figure 5.5 gives an example of this non-linear function of

the local bandwidth usage of *streamcluster* on an AMD six-core processor. Consequently, we describe the local bandwidth demand $L_{D,i}$ with a discrete function given in Equation (5.13), where $b_{c,i}$ represents the value of the local bandwidth demand when $c$ cores are used on $node_i$.

$$L_{D,i} = f(a_i) = \begin{cases} b_{0,i}, & \text{if } a_i = 0 \\ b_{1,i}, & \text{if } a_i = 1 \\ \vdots & \\ b_{n_i,i}, & \text{if } a_i = n_i \end{cases} \tag{5.13}$$

Unfortunately, discrete functions cannot be used as MIP constraints. We present a novel technique which converts discrete functions to linear functions. First, we convert Equation (5.13) into a linear function by introducing auxiliary variables $y_{c,i}$.

$$\begin{aligned} &L_{D,i} = y_{0,i} \cdot b_{0,i} + y_{1,i} \cdot (b_{1,i} - b_{0,i}) + \cdots + y_{n_i,i} \cdot (b_{n_i,i} - b_{n_i-1,i}), \\ &y_{c,i} \text{ is an } \textbf{integer}, 0 \leq y_{c,i} \leq 1, \\ &y_{c,i} = 1 \text{ if } a_i \geq c; \text{ else } y_{c,i} = 0; 0 \leq c \leq n_i \end{aligned} \tag{5.14}$$

When $a_i = c$, Equation (5.14) is $L_{D,i} = b_{0,i} + b_{1,i} - b_{0,i} + \cdots + b_{c,i} - b_{c-1,i} = b_{c,i}$, equivalent to Equation (5.13). Next, we use a common method to convert the $if-else$ condition in Equation (5.14) to linear constraints with Equation (5.15), where $B$ and $\epsilon$ are arbitrary constants, $0 < \epsilon < 1$, $B \gg n_i$ [83].

$$\begin{aligned} &B \cdot y_{c,i} \geq a_i - c + \epsilon, \\ &c \cdot y_{c,i} \leq a_i. \end{aligned} \tag{5.15}$$

When $a_i \geq c$, Equation (5.15) is essentially $(y_{c,i} > 0) \wedge (y_{c,i} \leq 1) \implies y_{c,i} = 1$. When $a_i < c$, Equation (5.15) is essentially $(y_{c,i} \geq 0) \wedge (y_{c,i} < 1) \implies y_{c,i} = 0$. In summary, Equations (5.14) and (5.15) convert the discrete local bandwidth demands of Equation (5.13) into linear constraints.

### 5.4.2 Objective Function

Predicting the optimal core allocation requires satisfying two goals: maximizing the total bandwidth usage and minimizing the core allocation. The first goal is expressed with Equation (5.10). The second goal can be described as

$$\text{minimize: } \sum_i a_i. \tag{5.16}$$

Because minimizing a function is similar to maximizing the negative of it, we combine Equation (5.10) and (5.16) into one objective function. To emphasize that our priority is maximizing bandwidth, we multiply the bandwidth goal by a constant $C$.

$$\text{maximize: } C \cdot (\sum_i L_i + \sum_{i,j} I_{ij}) - \sum_i a_i. \tag{5.17}$$

### 5.4.3 NuCore Summary

Equation (5.18) summarizes the linear constraints and objective function of NuCore, including the constraints from NuMem.

$$\begin{aligned}
&\text{maximize: } C \cdot (\sum_i L_i + \sum_{i,j} I_{ij}) - \sum_i a_i \\
&\text{subject to: } \forall i, j: \\
&\quad \text{constraint}_1\text{: } I_{r,ji} \le a_i \times I_{r,ji,solo}, \\
&\qquad\qquad I_{w,ji} \le n_j \times I_{w,ji,single}, \\
&\qquad\qquad I_{ji} = I_{ij,r} + I_{ij,w} + \sum_{l,k} I_{lk}, \forall\ node_l \text{ and } node_k \\
&\qquad\qquad \text{connected through link } j \to i., \\
&\quad \text{constraint}_2\text{: } I_{ji} \le UniMax_{ji}, \\
&\quad \text{constraint}_3\text{: } I_{ji} + I_{ij} \le BiMax_{ij}, \\
&\quad \text{constraint}_4\text{: } \sum_{j=1}^{M} I_{ij} + \beta_i \times L_{D,i} \le \alpha_i, \\
&\qquad\qquad \sum_{j=1}^{M} I_{ij} + L_i \le \alpha_i, \\
&\quad \text{constraint}_5\text{: } 0 \le a_i \le n_i, \\
&\quad \text{constraint}_6\text{: } L_{D,i} = \sum_{c=0}^{n_i} y_{c,i} \cdot (b_{c,i} - b_{c-1,i}), \\
&\qquad\qquad \forall c, 0 \le c \le n_i\text{:} \\
&\qquad\qquad\quad B \cdot y_{c,i} \ge a_i - c + \epsilon, \\
&\qquad\qquad\quad c \cdot y_{c,i} \le a_i \\
&\qquad\qquad\quad y_{c,i} \text{ is integer}, 0 \le y_{c,i} \le 1
\end{aligned} \tag{5.18}$$

### 5.4.4 Theoretical Complexity of NuCore

One of the major goals of NuCore is to predict the optimal core allocation with low cost. Here, we briefly discuss the theoretical complexity of NuCore. Recall that $N$ denotes the total number of nodes, and $n$ denotes the number of cores per node. The MIP problem of Equation (5.18) has $N^2$ variables for inter-node bandwidth, $N$ variables for local bandwidth and $N$ variables for core allocations. Additionally, there are $n \cdot N$ variables for $y_{c,i}$. Similarly, there are $O(N^2+nN)$ constraints. That is, there are $O(N^2 + nN)$ variables and constraints. Although the worst case complexity is exponential, using the branch-and-bound method, most MIP problems can be solved in polynomial time, i.e., the complexity is usually $O((N^2+nN)^c)$ [87, 119]. Section 5.5 shows that a typical NuCore problem instance can be solved in less than a second with state-of-the-art MIP solvers.

## 5.5 Experimental Evaluation

This section provides experimental evaluation of the accuracy and performance benefits of our models.

### 5.5.1 Platforms, Benchmarks, Methodology and Metrics

**Platforms** We evaluated our models on the two large-scale NUMA platforms shown in Chapter 3. The Intel platform has 32 cores on 4 nodes of Intel X7550 2.0GHz processors. Each processor has eight cores sharing 18MB L3 cache and 64GB memory. Each core also has 64KB split L1 cache and 256KB L2 cache. These nodes are fully connected as illustrated in Figure 3.2 on page 27. All inter-node links are Quick Path Interconnect 1.0 (QPI) [66]. Table 5.1a gives the values of the machine configurations for this platform. This platform runs Linux 3.8.0.

The AMD platform has 48 cores on 8 nodes. This machine has four AMD Opteron 6174 2.2GHz processors in four sockets. Each processor has two six-core nodes sharing 6MB L3 cache and 12GB memory. Each core has 128KB split L1 Cache and 256KB L2 cache (same as described in Section 4.5.1). Figure 3.1 illustrates the connection topology of this platform, which is non-uniform [6]. There are several two-hop links. All links are HyperTransport (HT) 3.0 [148]. However, because intra-socket and inter-socket links have different bit-widths, the inter-node links are not homogeneous. Table 5.1b gives the machine configurations for this platform. This platform runs

| Config | Value |
|---|---|
| $UniMax_{ij}$ | 10.8 Gb/s |
| $BiMax_{ij}$ | 17.5 Gb/s |
| $\alpha_i, \beta_i$ | 20.26 Gb/s, 0.24 |

(a) Platform with Intel nodes

| Config | Value | |
|---|---|---|
| $UniMax_{ij}$ | 1-hop intra-socket | 4.9 Gb/s |
| | 1-hop inter-socket | 2.1 Gb/s |
| | 2-hop | 1.8 Gb/s |
| $BiMax_{ij}$ | 1-hop intra-socket | 8.5 Gb/s |
| | 1-hop inter-socket | 4.1 Gb/s |
| | 2-hop | 3.6 Gb/s |
| $\alpha_i$, $\beta_i$ | 8.1 Gb/s, 0.40 | |

(b) Platform with AMD nodes

Table 5.1: The values of the machine configurations of the experiment platforms.

Linux 2.6.23.

**Benchmarks** We used PARSEC 2.1 benchmarks and NPB-OMP 3.3.1 benchmarks in our evaluation [15, 75]. We also used the BLAS matrix multiplication routine *dgemm* in the Intel Math Kernel Library (MKL) 11.1 and AMD Core Math Library (ACML) 5.3.1, because its wide application and high bandwidth usage [7, 18, 68]. The evaluation included 22 benchmarks, among which 7 are memory-intensive (listed in Table 5.2) and 15 are CPU-intensive. Three PARSEC benchmarks *bodytrack*, *dedup*, and *x264* are not used because they are I/O bound and have very limited memory bandwidth usages. Because of the limited bandwidth usages, these three benchmarks perform the best with all cores. For PARSEC, the "native" input sets are used. For NPB, the largest executable input sets, "C" or "D" are used. For *dgemm*, two $1.6K \times 1.6K$ matrices with random values are multiplied to fully exercise the memory system [5].

PARSEC and NPB benchmarks are compiled using GCC/GFortran 4.4.3 with the optimization flags following the default configurations of the benchmark suites. For *dgemm*, libraries compiled by Intel and AMD are used.

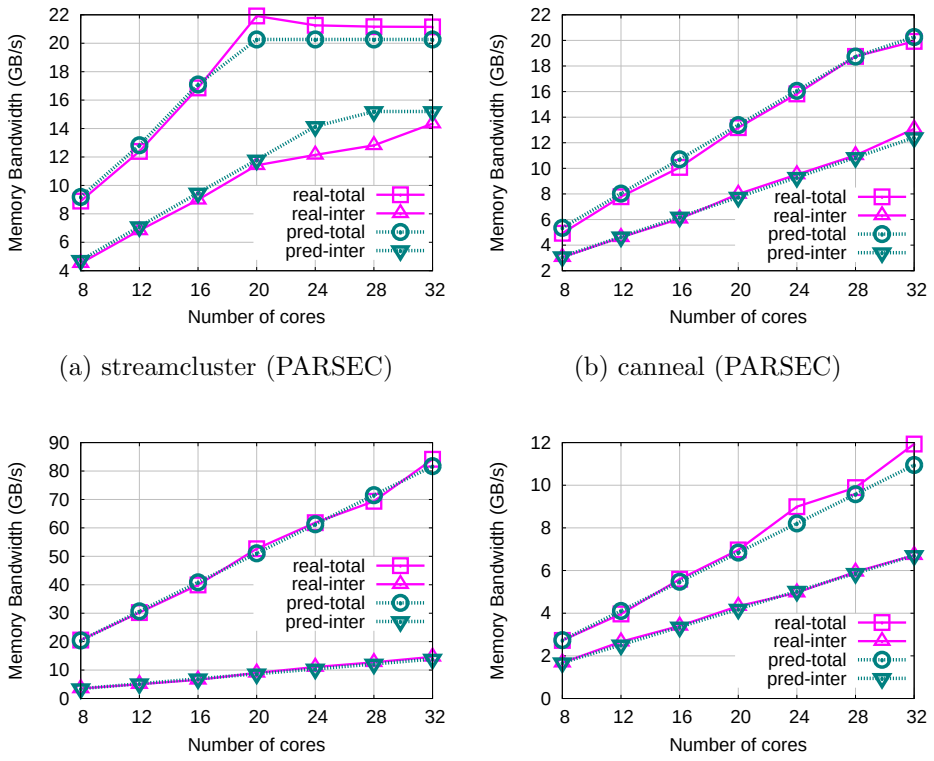
**Bandwidth Prediction Evaluation** As there are millions of possible core allocations, it is impractical to collect the actual bandwidth usages for all of them. Therefore, for each benchmark, we predicted its bandwidth usages using NuMem for ten randomly selected core allocations on each platform. We compare the predicted values and the actual values obtained from PMUs, and report the mean absolute

percentage error (MAPE) for each benchmark, which is defined as [64],

$$MAPE = \frac{1}{10} \sum_{alloc=1}^{10} \left| \frac{BW_{alloc,actual} - BW_{alloc,predicted}}{BW_{alloc,actual}} \right|. \tag{5.19}$$

**Core Allocation Prediction Evaluation** For each benchmark, we also predicted the optimal core allocation using NuCore. We compared the performance of the predicted core allocation with the use-all-cores allocation, and report the speedup of the predicted core allocation. Each performance result is the average from six trials. The speedup is defined as

$$SpeedUp = \frac{ExecTime_{use-all-cores}}{ExecTime_{predicted}}. \tag{5.20}$$




(a) streamcluster (PARSEC)

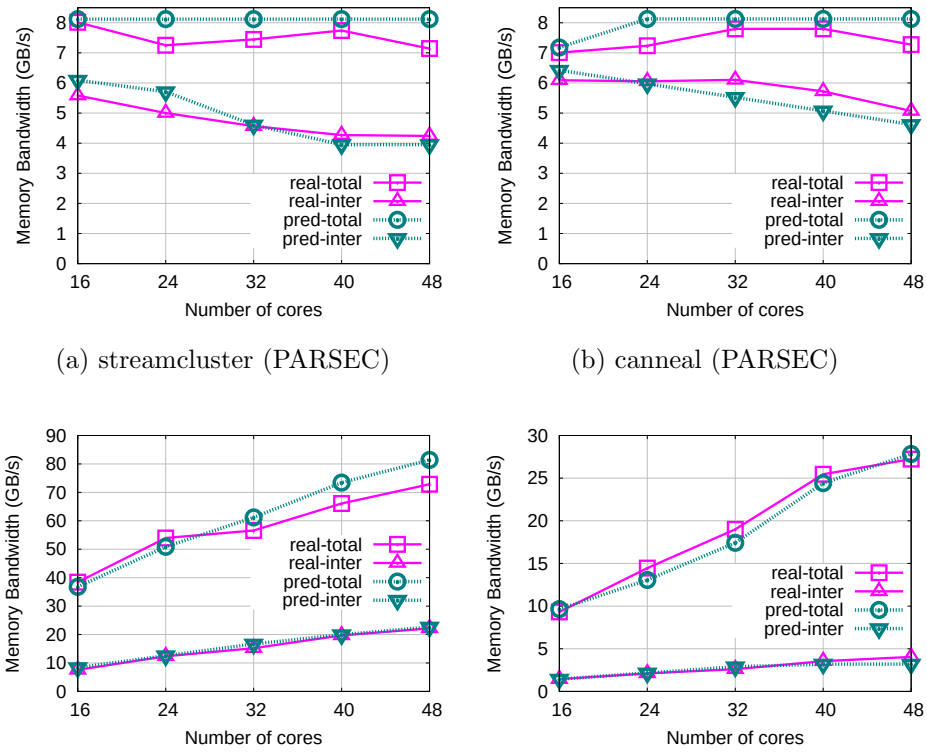
(b) canneal (PARSEC)

(c) mg.D-rprj3 (NPB)

(d) dgemm (MKL)

Figure 5.6: Prediction results of the total and inter-node bandwidth usages of four benchmarks on the Intel platform.

To verify the accuracy of the optimal core allocation prediction, we determined the optimal core allocation experimentally. We first picked the best performing core




(a) streamcluster (PARSEC)

(b) canneal (PARSEC)

(c) mg.D-rprj3 (NPB)

(d) dgemm (ACML)

Figure 5.7: Prediction results of the total and inter-node bandwidth usages of four benchmarks on the AMD platform.

allocation from a large pool of core allocations: 2000 randomly selected allocations, DraMon predicted optimal allocations, and some commonly-used core allocations (where each node is allocated with same number of cores, or linearly allocating cores from one core on $node_0$ to all cores on all nodes). We then determined the optimal core allocation experimentally by evaluating the performance of adjacent core allocations of this best allocation. Two core allocations $A$ and $A'$ are considered adjacent if they differ by only one core:

$$\begin{aligned} &A = \{a_0, a_1, \ldots, a_i, \ldots, a_N\} \text{ and } A' = \{a'_0, a'_1, \ldots, a'_i, \ldots, a'_N\} \\ &\text{are adjacent } \iff \{\exists! i, |a_i - a'_i| = 1\} \bigwedge \{\forall j, j \neq i, a_j = a'_j\} \end{aligned} \tag{5.21}$$

A core allocation is considered optimal if it performs better than all of its adjacent core allocations. We started with the predicted core allocation, and compare it with its adjacent allocations. If the performance of all adjacent allocations is worse than

| Benchmark | MAPE | | | | | |
|---|---|---|---|---|---|---|
| | Intel Platform | | | AMD Platform | | |
| | Local | Inter | Total | Local | Inter | Total |
| streamcluster | 8.8% | 5.8% | 4.0% | 16.1% | 6.4% | 8.0% |
| canneal | 10.2% | 2.4% | 3.3% | 12.9% | 6.9% | 6.4% |
| facesim | 8.3% | 6.8% | 9.7% | 3.3% | 4.8% | 2.1% |
| mg.D-resid | 8.2% | 0.0% | 8.2% | 7.8% | 0.0% | 7.8% |
| mg.D-psinv | 6.9% | 0.0% | 6.9% | 7.6% | 0.0% | 7.6% |
| mg.D-rprj3 | 1.9% | 4.3% | 2.0% | 10.9% | 4.9% | 8.2% |
| mg.D-interp | 10.5% | 6.7% | 7.5% | 9.8% | 6.9% | 8.4% |
| sp.C-x/y/zsovle | 10.1% | 8.3% | 5.1% | 10.4% | 5.4% | 8.7% |
| sp.C-rhs | 7.1% | 4.8% | 6.7% | 6.5% | 3.9% | 5.3% |
| dgemm (MKL) | 9.1% | 2.1% | 3.2% | 5.2% | 2.8% | 4.3% |
| dgemm (ACML) | 6.0% | 6.5% | 4.8% | 5.7% | 8.7% | 5.6% |
| Average | 7.9% | 4.3% | 5.6% | 8.7% | 4.7% | 6.7% |

Table 5.2: Average MAPE of bandwidth usage prediction for memory-intensive benchmarks.

the predicted core allocation, then the prediction is correct. Otherwise, we selected the adjacent core allocation with the best performance, and evaluate its adjacent allocations until we find the optimal one.

Note that the above experimental optimal core allocation can be local optimal. However, finding the global optimal from millions of core allocations experimentally is impractical. Instead, we compared the performance of the predicted allocations with the performance upper bound of all core allocations. If this upper bound has a tight confidence interval with high confidence, it can be viewed as the real optimal performance. We determined this upper bound with Extreme Value Theory (EVT) [14, 24, 126]. We sampled the performance of more than 2000 randomly selected core allocations as previous work [126]. Based on the distribution of the samples, the maximum value of the sample space, i.e., the performance upper bound, can be estimated with EVT. For the rest of this chapter, we call this upper bound the *real optimal performance*.

| Benchmark | NuCore Alloc (%) | Exp. Opt. Alloc (%) | Constraint |
|---|---|---|---|
| streamcluster | {8,8,3,0} (59.3%) | {8,8,2,0} (56.3%) | Max Out |
| canneal | {8,8,8,4} (87.5%) | {8,8,8,4} (87.5%) | Max Out |
| facesim | {8,8,0,0} (50.0%) | {8,8,0,0} (50.0%) | Max Out |
| mg.D-resid | {7,7,7,7} (87.5%) | {7,7,7,7} (87.5%) | Local BW |
| mg.D-psinv | {8,8,8,8} (100%) | {8,8,8,8} (100%) | None |
| mg.D-rprj3 | {8,8,8,8} (100%) | {8,8,8,8} (100%) | None |
| mg.D-interp | {8,8,8,8} (100%) | {8,8,8,8} (100%) | None |
| sp.C-x/y/zsovle | {3,3,3,3} (37.5%) | {3,3,3,3} (37.5%) | Local/Remote Cont. |
| sp.C-rhs | {8,8,8,8} (100%) | {8,8,8,8} (100%) | None |
| dgemm (MKL) | {8,8,8,8} (100%) | {8,8,8,8} (100%) | None |
| dgemm (ACML) | {8,8,8,8} (100%) | {8,8,8,8} (100%) | None |
| Average | 83.8% | 83.5% | |

| Benchmark | NuCore Spdup | Exp. Opt. Spdup | Real Opt. Spdup |
|---|---|---|---|
| streamcluster | 1.79 | 1.79 | 1.79 (1.79∼1.80) |
| canneal | 1.00 | 1.00 | 1.00 (1.00∼1.02) |
| facesim | 1.13 | 1.13 | 1.13 (1.13∼1.13) |
| mg.D-resid | 1.09 | 1.09 | 1.09 (1.09∼1.09) |
| mg.D-psinv | 1.00 | 1.00 | 1.01 (1.00∼1.01) |
| mg.D-rprj3 | 1.00 | 1.00 | 1.01 (1.00∼1.01) |
| mg.D-interp | 1.00 | 1.00 | 0.97 (0.97∼1.00) |
| sp.C-x/y/zsovle | 1.46 | 1.46 | 1.45 (1.44∼1.46) |
| sp.C-rhs | 1.00 | 1.00 | 1.00 (1.00∼1.00) |
| dgemm (MKL) | 1.00 | 1.00 | 1.00 (1.00∼1.02) |
| dgemm (ACML) | 1.00 | 1.00 | 1.00 (1.00∼1.01) |
| Average | 1.13 | 1.13 | 1.13 (1.13∼1.14) |

Table 5.3: Predicted and experimentally determined optimal core allocations for memory-intensive benchmarks on the Intel platform (speedup baseline is use-all-cores allocation)

### 5.5.2 Results for Bandwidth-limited benchmarks

#### Bandwidth Usage Prediction

Table 5.2 gives the average bandwidth prediction error for seven memory-intensive benchmarks. NPB benchmarks have several phases and the predictions are made for each phase. The function name of each phase is also supplied for NPB benchmarks in Table 5.2. Note that there are several cases where the benchmarks have no inter-node bandwidth usage, where NuMem has 0% inter-node bandwidth prediction error. To illustrate the prediction accuracy for individual core allocations, Figure 5.6 and 5.7 shows the bandwidth prediction results of several core allocations for four benchmarks (other benchmarks have similar results). The x-axes of both figures represent the numbers of cores of the allocations, which are evenly spread on the nodes. The figures and table show that NuMem is highly accurate for memory-intensive benchmarks.

The highest error on the Intel platform is 36.9% when predicting the local band-

| Benchmark | Varuna Alloc (%) | Varuna Spdup |
|---|---|---|
| streamcluster | {8,8,8,8} (100%) | 1.00 |
| canneal | {8,8,8,8} (100%) | 1.00 |
| facesim | {4,4,4,4} (50%) | 1.04 |
| mg.D-resid | {8,8,8,8} (100%) | 1.00 |
| mg.D-psinv | {8,8,8,8} (100%) | 1.00 |
| mg.D-rprj3 | {8,8,8,8} (100%) | 1.00 |
| mg.D-interp | {8,8,8,8} (100%) | 1.00 |
| sp.C-x/y/zsovle | {5,5,4,4} (78.3%) | 1.21 |
| sp.C-rhs | {8,8,8,8} (100%) | 1.00 |
| dgemm (MKL) | {8,8,8,8} (100%) | 1.00 |
| dgemm (ACML) | {8,8,8,8} (100%) | 1.00 |
| Average | 93.5% | 1.02 |

Table 5.4: Predicted optimal core allocations for memory-intensive benchmarks on the Intel platform by Varuna (speedup baseline is use-all-cores allocation) [138].

width usage of *streamcluster* with core allocation $\{8, 6, 3, 7\}$. On the AMD platform, the highest error is 30.0% when predicting the local bandwidth usage of *streamcluster* with core allocation $\{2, 2, 2, 2, 2, 2, 1, 1\}$. This high error is caused by the fluctuation of the PMU readings, which is in turn caused by *Streamcluster*'s short memory bursts. These bursts are too short to be stably caught by the PMUs [155]. This fluctuation affects both profiling as well as the acquired real bandwidth usages. A better PMU handling in the OS kernel can mitigate this problem [36].

Because it is impossible to evaluate NuMem for every core allocation, we only predicted ten randomly-picked allocations for each benchmark. With Student's test, we can show that this experiment design is statistically sound: these results show that NuMem's average error is lower than 10% for local, inter-node, and total bandwidth predictions on both platforms with 99% confidence.

### Optimal Core Allocation Prediction

Table 5.3 and Table 5.5 give the optimal core allocation (and the percentage of cores used) predicted by NuCore for seven memory-intensive benchmarks. It also gives the experimentally determined optimal core allocation ("Exp. Opt. Alloc"). Table 5.3 and Table 5.5 show that NuCore can correctly predict the optimal core allocation for most benchmarks. Only one benchmark, *streamcluster*, was mispredicted on the Intel platform. Four benchmark phases, *sp.C-rhs*, *mg.D-psinv*,*mg.D-rprj3* and *mg.D-interp* were mispredicted on the AMD platform. All these mispredicted core allocations differ only by one core per node with the experimentally determined optimal core allocations.

| Benchmark | NuCore Alloc (%) | Exp. Opt. Alloc (%) | Constraint |
|---|---|---|---|
| streamcluster | {6,0,0,0,0,0,0,0} (12.5%) | {6,0,0,0,0,0,0,0} (12.5%) | Max Out |
| canneal | {6,6,6,3,0,0,0,0} (43.8%) | {6,6,6,3,0,0,0,0} (43.8%) | Max Out |
| facesim | {6,6,4,0,0,0,0,0} (33.3%) | {6,6,4,0,0,0,0,0} (33.3%) | Max Out |
| mg.D-resid | {6,6,6,6,6,6,6,6} (100%) | {6,6,6,6,6,6,6,6} (100%) | None |
| mg.D-psinv | {5,5,5,5,5,5,5,5} (83.3%) | {6,6,6,6,6,6,6,6} (100%) | None |
| mg.D-rprj3 | {6,5,6,6,6,4,6,4} (89.6%) | {6,5,6,6,6,4,6,5} (91.7%) | Inter-Comm |
| mg.D-interp | {6,5,6,6,6,4,6,4} (89.6%) | {6,5,6,5,6,4,6,5} (91.7%) | Inter-Comm |
| sp.C-x/y/zsovle | {1,1,1,1,1,1,1,1} (16.7%) | {1,1,1,1,1,1,1,1} (16.7%) | Local/Remote Cont. |
| sp.C-rhs | {6,5,6,4,5,4,5,4} (81.3%) | {6,5,6,5,5,4,4,4} (81.3%) | Local BW |
| dgemm (MKL) | {2,6,6,6,6,6,6,6} (91.7%) | {2,6,6,6,6,6,6,6} (91.7%) | Local/Remote Cont. |
| dgemm (ACML) | {3,6,6,6,6,6,6,6} (93.8%) | {3,6,6,6,6,6,6,6} (93.8%) | Local/Remote Cont. |
| Average | 66.9% | 69.7% | |

| Benchmark | NuCore Spdup | Exp. Opt. Spdup | Real Opt. Spdup |
|---|---|---|---|
| streamcluster | 3.34 | 3.34 | 3.33 (3.33∼3.34) |
| canneal | 1.38 | 1.38 | 1.37 (1.37∼1.38) |
| facesim | 1.18 | 1.18 | 1.19 (1.19∼1.20) |
| mg.D-resid | 1.00 | 1.00 | 1.00 (1.00∼1.00) |
| mg.D-psinv | 0.93 | 1.00 | 0.99 (0.99∼1.00) |
| mg.D-rprj3 | 1.08 | 1.11 | 1.10 (1.10∼1.10) |
| mg.D-interp | 1.08 | 1.09 | 1.10 (1.10∼1.11) |
| sp.C-x/y/zsovle | 1.59 | 1.59 | 1.59 (1.59∼1.59) |
| sp.C-rhs | 1.24 | 1.30 | 1.30 (1.30∼1.30) |
| dgemm (MKL) | 1.55 | 1.55 | 1.55 (1.55∼1.56) |
| dgemm (ACML) | 1.15 | 1.15 | 1.15 (1.15∼1.16) |
| Average | 1.41 | 1.42 | 1.42 (1.42∼1.43) |

Table 5.5: Predicted and experimentally determined optimal core allocations for memory-intensive benchmarks on the AMD platform (speedup baseline is use-all-cores allocation)

| Benchmark | Varuna Alloc (%) | Varuna Spdup |
|---|---|---|
| streamcluster | 2,2,1,1,1,1,1,1 (20.8%) | 1.85 |
| canneal | {6,6,6,6,6,6,6,6} (100%) | 1.00 |
| facesim | {2,2,2,2,2,2,2,2} (33.3%) | 1.02 |
| mg.D-resid | {3,3,3,3,3,3,3,2} (47.9%) | 0.62 |
| mg.D-psinv | {6,6,6,6,6,6,6,6} (100%) | 1.00 |
| mg.D-rprj3 | {6,6,6,6,6,6,6,6} (100%) | 1.00 |
| mg.D-interp | {6,6,6,6,6,6,6,6} (100%) | 1.00 |
| sp.C-x/y/zsovle | {2,2,2,1,1,1,1,1} (22.0%) | 1.29 |
| sp.C-rhs | {4,4,4,3,3,3,3,3} (56.3%) | 1.14 |
| dgemm (MKL) | {6,6,6,6,6,6,6,6} (100%) | 1.00 |
| dgemm (ACML) | {6,6,6,6,6,6,6,6} (100%) | 1.00 |
| Average | 70.9% | 1.13 |

Table 5.6: Predicted optimal core allocations for memory-intensive benchmarks on the AMD platform by Varuna (speedup baseline is use-all-cores allocation) [138].

Table 5.3 and Table 5.5 also gives the primary bandwidth constraint for each benchmark in the "Constraint" column. In this column, "Local BW" refers to the benchmarks that are primarily local memory bandwidth limited as discussed in Section 3.2.1. "Max Out" refers to the benchmarks whose data are fully shared, and whose scalability is primarily limited by the maximum output bandwidth of the shared-data-node, as discussed in Section 3.2.3. "Local/Remote Cont." refers to the benchmarks whose data are partially shared, and whose scalability is primarily limited by the local and remote accesses on the shared-data-node, as discussed in Section 3.2.3. "Inter-Comm" refers to the benchmarks with inter-thread communication, and are limited by the inter-node memory bandwidth, as discussed in Section 3.2.2.

Table 5.3 and Table 5.5 also give the speedup of NuCore-predicted optimal core allocations ("NuCore Spdup") over use-all-core allocations. As the table shows, NuCore-predicted core allocation has a maximum speedup of 3.34 using only 12.5% cores when executing *streamcluster* on the AMD platform. On the Intel platform, the average speedup for the memory-intensive benchmarks is 1.13. On the AMD platform, the average speedup is 1.42. More importantly, these speedups are achieved with much fewer resources. NuCore only allocates 82.7% of all available cores on the Intel platform on average, and 67.3% of all available cores on the AMD platform. The overall average speedup is 1.27 with only 75.6% cores allocated.

We observe that there are two reasons for the slowdown of allocating more cores than optimal. The first reason is the increasing local-remote-access contention which slightly reduces the total bandwidth usages for certain benchmarks. The second reason is the data starvation caused by bandwidth over-saturation. We observe that

some threads have lower share of bandwidth than the others. These threads become bottlenecks and slow down the whole program.

Table 5.3 and Table 5.5 also show that NuCore's core allocations perform closely to the experimentally-determined optimal core allocations ("Exp. Opt. Spdup"). The average performance difference between the two is only 0.5%. Table 5.3 and Table 5.5 also give the real optimal performance (and its confidence interval with 95% confidence) estimated using EVT ("Real Opt. Spdup"). The average performance difference between real optimal and DraMon is only 1.0%.

Table 5.4 and Table 5.6 give the optimal core allocations predicted by the state-of-art predictive technique, Varuna, along with the performance of these predicted core allocations [138]. Varuna treats each node similarly. It only predicts the optimal number of cores, and allocates cores evenly from each node based on the predicted core count. As Table 5.4 and Table 5.6 show, Varuna mispredicted the optimal numbers of cores for four phases on the Intel platform and seven phases on the AMD platform. These results suggesting that the accuracy of Varuna is considerable lower than NuCore, which only mispredicted one phase on the Intel platform and four phases on the AMD platform. Additionally, the core allocations predicted by Varuna performed 19% worse than the those predicted by NuCore on average. These results indicates the importance of considering NUMA heterogeneity when predicting the optimal core allocation.

### 5.5.3 Results for Not-bandwidth-limited Benchmarks

Although the primary goal of NuCore is to correctly predict the optimal core allocation for bandwidth-limited applications, it is also very important that NuCore does not mispredict a not-bandwidth-limited application as bandwidth limited. For the rest of the benchmarks, NuCore correctly predicted that they were not memory bandwidth limited. That is, if no other significant scalability issues exist, these benchmarks perform best using all cores. On our experiment platforms, these benchmarks indeed perform best when using all cores (other scalability issues are discussed in Section 5.6).

NuMem can also predict the bandwidth usages of non-bandwidth-limited benchmarks with high accuracy. The average MAPE for local, inter-node and total bandwidth predictions for all 22 benchmarks on the Intel platform are 11.8%, 6.0%, and 8.8%, respectively. The corresponding errors on the AMD platform are 13.2%, 6.0%,

| Nodes | Cores/Node | Total Core # | Pred. Time (sec) | Notes |
|---|---|---|---|---|
| 4 | 8 | 32 | 0.02 | the Intel platform |
| 8 | 6 | 48 | 0.02 | the AMD platform |
| 8 | 128 | 1,024 | 0.17 | hypothetical |
| 10 | 128 | 1,280 | 0.21 | hypothetical |
| 100 | 128 | 12,800 | 21.85 | hypothetical |
| 100 | 512 | 51,200 | 237.93 | hypothetical |

Table 5.7: DraMon prediction time.

and 11.7%.

### 5.5.4 Prediction Time of NuCore

We used a state-of-the-art MIP solver, SCIP, to solve NuCore instances [2]. The maximum time to solve a NuCore instance on both Intel and AMD platforms is 0.02 seconds. The profiling phase requires running a program with one thread/core per node for 0.5 seconds to sample PMU readings. Therefore, the total prediction time is 0.52 seconds, which adds 0.5% overhead to the execution time of our benchmarks on average. This low overhead makes NuCore suitable for run-time optimization. To understand NuCore's performance for future large NUMA systems, we used DraMon to make predictions for several hypothetical platforms. The results are summarized in Table 5.7. Table 5.7 shows that NuCore's prediction time remains low for extremely large systems. Even in the case of a 100-node system with total 50K cores, a NuCore instance can be solved in 237.93 seconds. We believe this is a reasonable time for a large and long-running program that can utilize 50K cores.

## 5.6 Discussion

**Impact of Program Parallelism, Cache Contention and Synchronization on Core Allocation:** The primary goal of NuCore is predicting the optimal core allocations for bandwidth-limited programs. Other scalability limitations, such as program parallelism, cache contention and synchronization are beyond the scope of this dissertation. Because of their distinct characteristics, different scalability limitations should be modeled individually. Existing or new techniques that focused on other scalability limitations can be used with NuCore to predict optimal core allocations to all types of programs [25, 60, 79, 86, 123, 144].

Additionally, the only scalability limitation that is significant enough to affect core allocation decisions on our systems is memory bandwidth. Because we used large input sets designed for evaluating large systems, our benchmarks have abundant parallelism. We did observe that synchronization and cache contention affect scalability. However, because of the large caches (18MB and 6MB), the fast inter-node links and large input sets, the impact of synchronization and cache contention is not significant enough to affect core allocation decisions on our systems. For systems having smaller caches, slower inter-node links or smaller workloads, parallelism, synchronization and cache contention may be significant. However, as stated above, they should be modeled separately.

**Cache Impact on Bandwidth Prediction:** For the seven bandwidth-limited benchmarks which already have high cache miss rates, their memory behaviors are not affected significantly by cache contention.[1] Therefore, NuMem achieves high accuracy for these benchmarks without a cache model.

Most of the not-bandwidth-limited benchmarks have very low cache miss rates, and are also not affected significantly by the cache contention.[1] As a results, NuCore also works well with most not-bandwidth-limited benchmarks. There are two benchmarks, *freqmine*, and *ep.D*, that are more sensitive to the contention and data sharing in the cache than other benchmarks. Including a cache model can improve the accuracy of the bandwidth usage prediction for them [45, 60, 145, 160].

**Prefetcher Impact:** The prefetchers of the AMD platform are enabled. AMD prefetchers are less-aggressive and can reduce prefetching requests in case of contention [6]. Therefore, these prefetchers do not affect our models' accuracy.

The Intel prefetchers are more aggressive, and consume considerable amount of memory bandwidth and cache space [44, 121]. Without detailed knowledge of the proprietary prefetching algorithm, it is extremely difficult to predict the prefetcher's impact. Therefore, we disabled the prefetchers on the Intel platform. Note that NuCore can still improve the performance of many benchmarks and correctly predict the optimal core allocations of many benchmarks with Intel prefetchers enabled. We plan to model Intel's prefetcher in the future.

[1] A similar observation was made by previous research [161, 165].

## 5.7 Summary

This chapter presented two models, NuMem and NuCore, to predict the bandwidth usages and optimal core allocation for multi-threaded programs. Both models convert the prediction problems into Mixed Integer Programming (MIP) problems to make predictions. When evaluated on two large scale NUMA machines, NuMem shows a low average prediction error of 10% for bandwidth usage predictions. The optimal core allocations predicted by NuCore provides 1.27 speedup with over use-all-cores allocations on average, with only 75.6% of all available cores allocated. The use of MIP ensures that NuCore can make predictions within 0.02 seconds.

# Chapter 6

# FlexThread: A Low-overhead and Efficient Run-time Thread Manager

## 6.1 Introduction

Chapter 4 and Chapter 5 presented the models for predicting the optimal core allocations for multi-threaded applications on large-scale NUMA platforms. However, after the online prediction of the optimal core allocations, multi-threaded applications have to adapt to their changing optimal core allocations during execution, which is still a challenge to overcome.

As stated in the Chapter 1, the rigidity of current thread libraries prevents multi-threaded applications from being reconfigured to use their optimal core allocations at run-time without incurring a high performance penalty. Recall the example of a multi-threaded application running on a machine with 16 cores. Initially, because the optimal core allocation is unknown, the application is executed using all 16 cores with 16 threads. After brief profiling and model calculation, the application is predicted to perform best with only 10 cores. However, because of the rigidity of current applications, it is impossible to change the number of threads during execution. Therefore, we have to execute 16 threads on 10 cores, causing an unbalanced load on the cores and performance degradation.

The inability to change the thread count of an application restricts previous optimal core allocation research to applications with data-parallel loops [32, 33, 60, 144].

The techniques that aim at enabling run-time core reallocation for any type of applications, either require source code modification or incur high overhead on NUMA machines. Hence, these techniques are not readily applicable to existing applications or large-scale NUMA platforms.

This chapter presents a run-time technique, FlexThread, which enables the core reallocation for any multi-threaded application during execution, without high performance penalty. FlexThread makes use of the fact that current multi-threaded applications can partition their work into fine-grained small jobs when they are instructed to create and utilize a large number of threads, with one thread assigned to process one small job. When a multi-threaded application adapts to a new core allocation, FlexThread redistributes all jobs to this new core allocation in a way that ensures each core gets similar load (similar number of jobs). Because one thread is responsible for one job, redistributing a job to a new core is equivalent to re-mapping its corresponding thread to the new core. The downside of creating numerous threads is that it increases the number of synchronization operations. This increase can introduce significant overhead. To reduce this overhead, FlexThread employs distributed synchronization primitives.

We evaluated FlexThread on our two large-scale NUMA platforms (described in Section 5.5.1) using PARSEC and NPB benchmarks [15, 75]. The experiment results show that, compared to executing these benchmarks using one thread per core with their optimal core allocations, FlexThread has less than 5% overhead. This low overhead suggests that FlexThread can efficiently support the execution of multi-threaded applications with varying optimal core allocations. Moreover, our results show that, by improving load balancing, data sharing and processor utilization, FlexThread can improve the performance of multi-threaded applications (over using just one thread per core) by up to 382.9%. This performance improvement suggests that FlexThread is actually a better option for executing multi-threaded applications.

The rest of this chapter is organized as follows. Section 6.2 discusses the load-balancing problem in depth and presents our solution to this problem. Section 6.3 analyzes the potential overhead of our solution and discusses how to mitigate this overhead. Section 6.6 analyzes how to determine the exact number of threads to created. Section 6.7 presents the implementation of FlexThread. Section 6.8 evaluates the performance and overhead of FlexThread, and Section 6.9 summarizes this chapter.

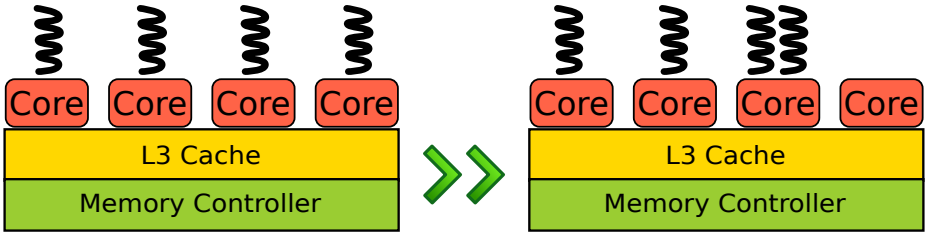



Figure 6.1: An example of a multi-threaded application starts with four threads on four cores, which is adapted to use three-core optimal allocation during execution.

## 6.2 Solving Load-balancing Problem

This section quantitatively evaluates the performance penalty of current run-time core reallocation techniques, and provides a solution to mitigate this performance penalty.

### 6.2.1 The Load-Balancing Problem

As discussed in Section 6.1, the inability to change thread count during execution causes unbalanced core loads among cores when adapting from one core allocation to another. This unbalanced load can incur a high performance penalty. Figure 6.1 gives an example to illustrate the severity of this performance penalty. Note that although the example in Figure 6.1 utilizes only one node of a NUMA machine, the problem is universal and exists in use cases with multiple nodes. In Figure 6.1, a multi-threaded application is executed on a quad-core processor. Because the optimal core allocation of this application is unknown at the start-up time, four threads are created to utilize all four cores. During execution, it is predicted that using three cores is the best core allocation. Therefore, the application has to be adapted to use three cores. However, because the thread count cannot be changed during execution, the application is forced to execute four threads on three cores, which implies that one core has to execute two threads, causing an unbalanced load. To estimate the performance penalty caused by this unbalanced load, assume the sequential (single-thread) execution time of this application is $T$. Ideally, when the application executes on three cores, only three threads have to be created. As a result, the ideal execution time on three cores is $\frac{T}{3}$ (assuming even job partitioning). However, in Figure 6.1, four threads are running on three cores, with two threads running on one core. Consequently, the execution time in Figure 6.1 is $\frac{2T}{4}$, which is 52% slower than the ideal execution time of $\frac{T}{3}$. That is, because of the rigidity of thread count, there is a 52%

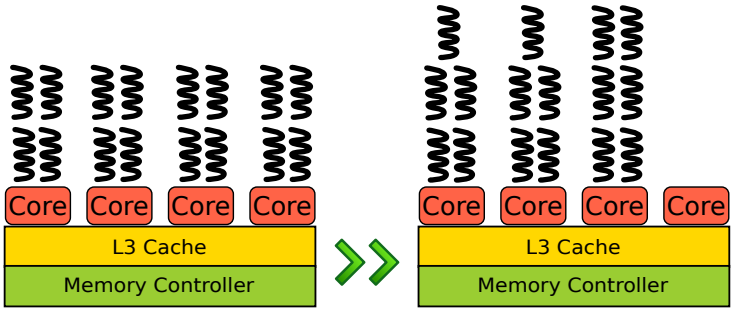



Figure 6.2: An example of a multi-threaded application starts with sixteen threads on four cores, which is adapted to use three-core optimal allocation during execution.

performance penalty when adapting from one core allocation to another, for this example. Similarly, on a NUMA machine with 32 cores, this performance penalty can be as high as 75% in theory.[1]

### 6.2.2 Solution to the Load-Balancing Problem

Although many multi-threaded applications do not allow changing thread count during execution, nearly all multi-threaded applications allow users to specify the numbers of threads to use for execution. Based on the numbers of threads specified, the algorithms of these applications partition their work (roughly) evenly into equal numbers of small jobs, and assign one job to one thread. Consequently, if an application is instructed to create a large number of jobs, it can partition its work into fine-grained small jobs. These fine-grained small jobs can easily be mapped to a allocation to achieve near-ideal load balancing.

Figure 6.2 presents an example similar to the case in Figure 6.1. However, unlike Figure 6.1, sixteen threads are created in Figure 6.2 instead of four. In Figure 6.2, when executing sixteen threads on three cores, two cores are assigned with five threads, while one core is assigned with six threads. Similar to the example in Figure 6.2, let the sequential execution time of this application be $T$. The execution time of sixteen threads on three cores is then $\frac{6T}{16}$, which is 14% slower than the ideal execution time of using only three threads. This slowdown is considerably smaller than the 52% slowdown in Figure 6.1. If more threads can be created, this slowdown can be further reduced. For example, if 128 threads are created, then the execution time is $\frac{43T}{128}$, which is only 2% slower than using three threads.

In summary, When an application creates a large number of threads, the applica-

[1]This 75% overhead happens when executing 32 threads on an optimal core allocation of 28 cores

tion partitions its work into fine-grained small jobs. These small jobs can be easily mapped to any core allocation with near-ideal load balancing, and thus allows the application to adapt to any core allocation without significant performance penalty from unbalanced loads.

## 6.3 Mitigating Overhead

Creating large numbers of threads can reduce the performance penalty from an unbalanced load. However, executing large numbers of threads may also introduce considerable overhead. This section analyzes the source of this overhead, and proposes solutions to mitigate this overhead.

## 6.4 The Overhead of Massive Threads

Although executing large numbers of threads solves the problem of unbalanced loads, it increases the execution time of an application because it increases the number of context switches and synchronization operations and thus increases the overhead.

To understand the impact of additional context switches and synchronizations, we experimented with a synthetic benchmark called *sync* on the 32-core Intel NUMA machine described in Section 5.5. *Sync* creates several threads. Each thread repeatedly multiplies an integer with itself, and saves the result back to this integer. Because each thread only uses one integer, there is little use of memory, allowing us to focus on the overhead of context switches and synchronizations. The benchmark stops when a total of $1.3 \times 10^{11}$ multiplications are performed by all threads, as $1.3 \times 10^{11}$ multiplications represent a workload large enough to fully utilize 32 cores.

To determine the overhead introduced by context switches, we ran the benchmark with two configurations. In the first configuration, or the "base-line" configuration, the benchmark is executed using different numbers of cores with thread counts always equal to the number of cores. In the second configuration, the benchmark is executed using different numbers of cores with 256 threads. By comparing these two configurations, we can determine the overhead from the context switches of large numbers of threads. Figure 6.3 gives the execution times of these two configurations. As Figure 6.3 shows, both configurations perform similarly, indicating that context switches add little overhead. Using lmBench benchmarks, we also determined the time of a context switch, which is 2000ns [98]. On Linux, the scheduling time slice is 100ms

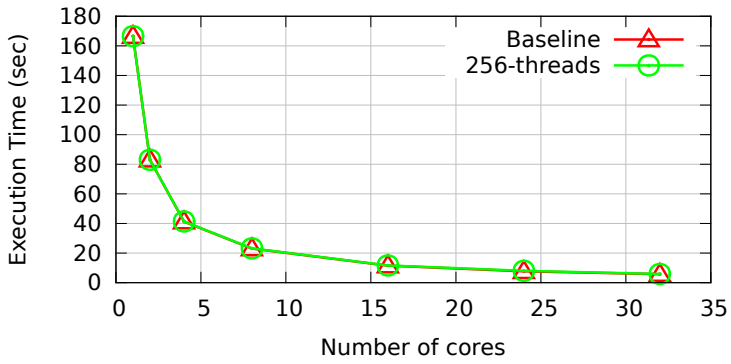



Figure 6.3: The execution time of *sync* under two configurations. "Baseline": same numbers of threads and cores (1 thread per cores). "256-threads": always use 256 threads. Only context switch overhead is included in this experiment.

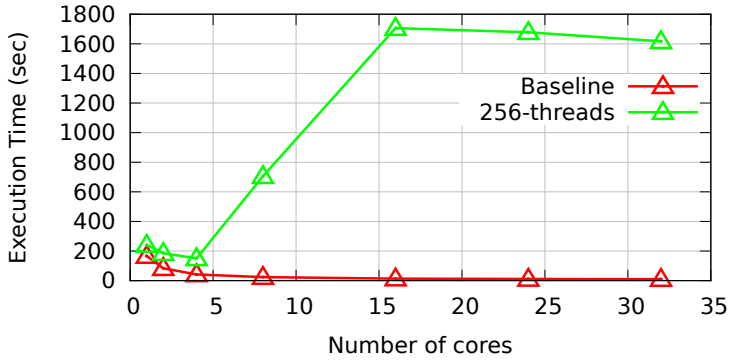



Figure 6.4: The execution time of *sync* under two configurations. "Baseline": same numbers of threads and cores (1 thread per cores). "256-threads": always use 256 threads. Synchronization overhead is included in this experiment.

or $1 \times 10^8$ns. That is, for every 100ms of execution, an extra 2000ns context switch time is required. Comparing to the time slice, 2000ns is negligible, suggesting that context switches add little overhead.

To determine the overhead introduced by synchronization operations, we extended *sync* with barriers: after every $1.3 \times 10^6$ multiplications, each thread of *sync* pauses, and calls *pthread_barrier* to synchronize with each other. We choose to stop every $1.3 \times 10^6$ multiplications to mimic the synchronization frequency of PARSEC benchmark *streamcluster*, which is one of the benchmarks with the highest synchronization frequency [15]. Pausing every $1.3 \times 10^6$ multiplications roughly equals pausing every 0.0015 seconds. The benchmark still stops when a total of $1.3 \times 10^{11}$ multiplications are performed. We ran the benchmark with two configurations. In the first config-

uration, or the "base-line" configuration, the benchmark is executed using different numbers of cores with a thread count that equals the core count. In the second configuration, the benchmark is executed using different numbers of cores with 256 threads. Figure 6.4 gives the execution times of these two configurations. As Figure 6.4 shows, synchronization operations add significant overhead when executing large numbers of threads. When executing 256 threads on 32 cores, the overhead is as high as 16670.5%. Similarly, we observe that a conditional variable, which is another common type of synchronization, also adds considerable overhead.

In summary, our experimental results demonstrate that synchronization operations are the primary cause of overhead when executing large number of threads on NUMA machines.

## 6.5 Mitigating Synchronization Overhead

The performance impact of large-scale parallel systems has long been studied and understood [3, 9, 46, 54, 56, 97, 99, 122, 124, 131, 132, 156]. The fundamental cause of the synchronization overhead is the sharing of the synchronization variables among nodes. These variables are used by concurrent threads to record their execution status and control the access to common resources. For example, in a barrier, one integer is shared by concurrent threads to record how many threads have reached the barrier. For correctness, these variables must be updated with atomic operations. On a NUMA machine, an atomic update is very expensive when the variable is shared by threads on multiple nodes, because there are usually three steps involved in an atomic update. First, the node that has the newest value of a shared variable must send this value to the node who wants to make an update. Second, the node that wants to make an update must send messages to the other nodes to invalidate their cached copies of this shared variable. In the last step, the value of the shared variable is updated. When there are large numbers of threads which all want to update a shared variable, the value of the share variable and the invalidation messages are frequently sent back and froth among nodes, which greatly increases the execution time.

Previous research solved the performance bottleneck of the shared variable by decomposing one shared global variable into multiple local variables, and utilize a hierarchical design of local and global variables [3, 9, 46, 54, 56, 97, 99, 122, 124, 131, 132, 156]. Threads running on a node mostly access the variable that is local to their node. Threads only access a global variable when they have to communicate

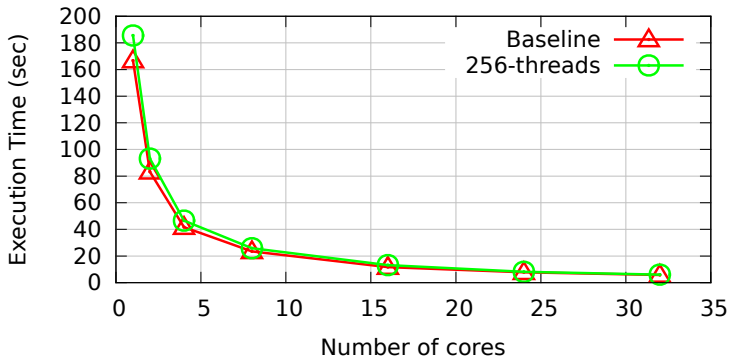



Figure 6.5: The execution time of *sync* with distributed synchronizations under two configurations. "Baseline": same numbers of threads and cores (1 thread per cores). "256-threads": always use 256 threads.

with threads on other nodes. In this way, unnecessary inter-node communications are eliminated. For example, the integer counter of a barrier can be decomposed into several node-level counters and one global counter. When a thread reaches the barrier, it first updates the local counter of its node. If there are still other threads on this node that have not reached the barrier, this thread simply goes to sleep. There is no inter-node communication required for this case. If all threads on this node have reached the barrier, then this thread can proceed to update the global counter. Only one global variable atomic update is required for each node. As a result, the overhead of synchronization can be mitigated.

To determine how much overhead can be mitigated by distributing shared variables, we applied the distributed synchronization algorithms designed by Mellor-Crummey and Scott to our synthetic benchmark *sync* [99, 132]. We then ran the new *sync* benchmark using the same two configurations as those found in Figure 6.4. The experimental results are shown in Figure 6.5. As Figure 6.5 shows, distributing shared variables can significantly reduce the overhead of synchronizing large numbers of threads. In Figure 6.5, the overhead of executing 256 threads on 32 cores is only 4.5%, compared to the 16670.5% overhead of non-distributed synchronization.

In summary, distributed synchronization primitives can significantly reduce the overhead of executing large numbers of threads. Combined with the benefit of better load balancing from executing large numbers of threads, it is possible to adapt a multi-threaded application to any core allocation during execution without high performance penalty.

## 6.6 How Many Threads to Create?

If we examine the results in Figure 6.5 closely, we can see that although the overhead of running 256 threads on 32 cores is low, the overhead (45%) of running 256 threads on 1 core is too high. It is worth noting that despite distributed synchronizations removing unnecessary inter-node communications, a synchronization operation still requires an atomic memory update and a system call to suspend a thread (in case of waiting). This update and system call cannot be eliminated. Consequently, there is still overhead associated with distributed synchronization when executing large numbers of threads. If the thread count is high, the overhead may still be unacceptable. [2] That is, there is a limit on how many threads can be created to ensure that the overhead remains within an acceptable level.

However, it is easy to see that, for load-balancing, the more threads created, the lower the performance penalty. Clearly, there is a trade-off between load-balancing and synchronization overhead, and it is important to determine the number of threads to create to ensure the overall slowdown is low. This section provides an analysis to determine this thread count, from both theoretical and practical perspectives.

### 6.6.1 Minimum Thread Count for Good Load-Balancing

For the slowdown caused by an unbalanced load, it is possible to determine the minimal thread count that is required to ensure a particular slowdown threshold by comparing the theoretical ideal performance with the theoretical performance of certain thread counts. The following paragraphs present this theoretical analysis.

Let the sequential execution time of an application be $T$. Let the core count of the optimal core allocation be $c$. The theoretically ideal performance is achieve when running $c$ threads on $c$ cores. That is, the ideal performance is

$$Perf_{ideal} = \frac{T}{c}. \tag{6.1}$$

Let the number of threads created be $t$. Running $t$ threads on $c$ implies that the highest number of threads per core is

$$\lceil \frac{t}{c} \rceil. \tag{6.2}$$

[2] Strictly speaking, if the thread count per core is high, the overhead may still be unacceptable. Given that the memory update and system call are mostly local to a core, thread count per core is more important than the total number of threads.

The performance of executing $t$ threads on $c$ cores is bound by the performance of the cores with the highest thread counts. Therefore, the performance of $t$ threads on $c$ cores is

$$Perf_t = \frac{\lceil \frac{t}{c} \rceil \times T}{t}. \tag{6.3}$$

Therefore, the slowdown of $t$ threads on $c$ cores, comparing to the ideal performance is

$$\frac{\lceil \frac{t}{c} \rceil \times c}{t} - 100\%. \tag{6.4}$$

Let the maximum number of cores on a NUMA machine be $n$. The core count, $c$, of an optimal core allocation can be any number between 1 and $n$. Let the maximum slowdown of running $t$ threads on any number of $c$ cores be $s_{max}$. $s_{max}$ can be computed easily as

$$s_{max} = max(\frac{\lceil \frac{t}{c} \rceil \times c}{t} - 100\%), 1 \leq c \leq n \tag{6.5}$$

If the maximum number of cores of a NUMA machine is known, it is easy to determine the maximum slowdown for different thread counts. For example, Table 6.1 gives the maximum slowdown for several thread counts on the 32-core Intel machine. If a desired slowdown is less than 5%, then based on Table 6.1, a minimum of 768 threads is required.

| Thread count | 128 | 256 | 512 | 768 | 1024 |
|---|---|---|---|---|---|
| Max slowdown | 32% | 15% | 8% | 5% | 3% |

Table 6.1: The theoretical maximum slowdown of executing various thread counts on the 32-core Intel platform.

However, based on our experimental results with PARSEC and NPB benchmarks, to ensure that slowdown is always smaller than 5%, only 256 threads are required in practice. Equation (6.5) is based on the assumptions that there is no interference from external sources, such as OS scheduling and memory contention. However, our experimental results show that external interference can cause at least 3% fluctuation. That is, the actual performance of executing one thread per core is at least 3% worse than the theoretical one-thread-per-core performance. Therefore, the actual slowdown of executing large number of threads than real one-thread-per-core execution is usually smaller than the theoretical values in Table 6.1. Additionally, the thread count difference between two cores is at most one. For 256 threads, one thread

only accounts for 0.4% of the total work of an application. Given that one thread represents only a small piece of work, the performance difference among cores is not significant with 256 threads. To sum up, in practice 256 threads is usually enough to ensure a maximum 5% slowdown on a 32 core machine.

In summary, Equation (6.5) provides a start point to select a few thread counts as candidates of the minimum thread count for their performance goals. Then experiments with a few benchmarks can determine which thread count is the actual minimum requirement. We provide tools to automate this procedure to help users determine the minimum thread count for their particular platforms and needs.

### 6.6.2 Maximum Thread Count for Low Synchronization Overhead

For the slowdown caused by synchronization, it is possible to determine the maximum thread count required to ensure a particular slowdown threshold, by computing the overhead from synchronization operations. The following paragraphs present the theoretical analysis.

As stated in the beginning of this section, a synchronization operation requires an atomic memory update and a system call to suspend a thread (in case of waiting). Because the memory update is local to a core, and the system calls on different cores can be parallelized, analyzing the overhead of synchronization operations is equivalent to analyzing the overhead of memory update and system call on a single core.

Let the core count of the optimal core allocation be $c$. Let the number of threads created be $t$. Equation (6.2) gives the thread count per core. Let the maximum possible synchronization frequency of an application be $f$ (synchronizations per second), when the application executes sequentially. Then the frequency of executing $t$ threads on $c$ cores is,

$$\lceil \frac{t}{c} \rceil \times f. \tag{6.6}$$

Let the time required for an memory update be $T_{mem}$, and the time required for a thread-suspending system call be $T_{syscall}$. Combining the frequency in Equation (6.6), the time that it takes to carry out synchronization operations per core for every second is

$$\lceil \frac{t}{c} \rceil \times f \times (T_{mem} + T_{syscall}). \tag{6.7}$$

Let the maximum acceptable slowdown be $s$. The maximum acceptable time for

synchronization per second is then $s \times 1$ *second*. The maximum thread count, denoted by $t_{max}$, that allows no more than $s \times 1$ *second* synchronization time every second is then

$$t_{max} = max(t | \forall c, (\lceil \frac{t}{c} \rceil \times f \times (T_{mem} + T_{syscall})) < s). \quad (6.8)$$

Equation (6.8) provides a means to estimate the maximum number of threads to satisfy a particular performance target, given that $s$, $f$, $T_{mem}$, $T_{syscall}$ and the range of $c$ are known. As an example, we compute the $t_{max}$ on the 32-core Intel platform. The maximum frequency, $f$, of our benchmarks on this platform is 667 per second (the frequency of *streamcluster*). Using lmBench, we can also determine that $T_{mem}$ and $T_{syscall}$ are 200ns and 2000ns. The maximum value of $c$ is the total number of cores on the Intel platform, which is 32. The minimum value of $c$, determined using the most memory-intensive benchmark *ld_mem* from lmBench, is 8.[3] Suppose the performance goal, $s$, is 5%, we can then compute the value of $t_{max}$, which is 272 threads. We provide tools to automate this procedure to help users determine the maximum thread count for their particular platforms and needs.

### 6.6.3 Summary for Thread Count Analysis

Section 6.6.1 and Section 6.6.2 provide an analysis to determine the maximum and minimum of the number of threads to create, so that a particular performance goal can be achieved for run-time core reallocation. Users can then select a number between the maximum and the minimum as the thread count for their applications. For example, on the 32-core Intel platform, the maximum number is 272, the minimum number is 256, and an user can choose 256 as their thread count. Because the slowdown imposed by unbalanced load and synchronization overhead is platform specific, determining the proper thread count requires experiments on the target platform. We also provide automatic tools to help users determine these values on their platforms.

## 6.7 Implementation

Incorporating the techniques of Section 6.2 and Section 6.3, we implemented the FlexThread technique to enable efficient run-time core reallocation. FlexThread provides distributed synchronization primitives that implement the algorithms designed

[3] *ld_mem* is a benchmark that is used to acquired the maximum bandwidth. Therefore, it is the most memory-intensive application we have.

by Mellor-Crummey and Scott [99, 132]. More specifically, FlexThread implements tree barriers, distributed mutexes and distributed condition variables. Because these algorithms are well documented by previous research, we do not elaborate on them in this dissertation.

However, because existing applications are hard-coded to utilized traditional non-distributed synchronization primitives, additional effort must be taken to convert these applications to distributed synchronizations without modifying their source code. When designing FlexThread, we particularlly focused on enabling distributed synchronizations for applications using POSIX Threads (Pthreads) or GNU OpenMP (GOMP), which are two popular thread libraries [65, 146].

FlexThread provides synchronization interfaces that are compatible with the POSIX Threads specifications. For example, FlexThread provides implementations for *pthread_barrier_wait* function, following POSIX interface standards. When executing an application, users must specify the value of an environment variable, "LD_PRELOAD", to be the path of the REEact library with FlexThread implementation. The OS then automatically links any Pthreads functions to the implementation of FlexThread. For example, when an application calls function *pthread_barrier_wait*, it invokes the distributed implementation of FlexThread instead of the default non-distributed implementation. In this way, all synchronization calls by multi-threaded applications are converted to distributed synchronizations without source code modification.

For GOMP, we acquired the source code of GOMP library, and modified GOMP source code to use distributed synchronizations. Note that GOMP source code is not application source code. It is the source code of the thread library used by multi-threaded applications. After GOMP library is converted to distributed synchronization, applications using GOMP library are automatically converted to distributed synchronizations without application source code modification.

## 6.8 Experimental Evaluation

This section presents the experimental evaluation of FlexThread's performance.

### 6.8.1 Experiment Setup

We evaluated FlexThread on our Intel and AMD NUMA platforms. A detailed description of the Intel and AMD platforms can be found in Section 5.5.1. We conducted

experiments with benchmarks from PARSEC and NPB benchmark suites [15, 75]. Two PARSEC benchmarks, *raytrace* and *x264*, experienced segmentation faults when executed with more than 64 threads, so they were excluded from our experiments. PARSEC's native input sets were used for PARSEC benchmarks, except for *swaptions*. For *swaptions*, we used a input set that was twice the size of its native input set. NPB's *D* input sets were used for NPB benchmarks. These input sets are large enough to fully utilize the large number of cores and threads used in our experiments. NPB benchmark *dc* was excluded from our experiments because it does not have *D* input set. The information of compiler and compilation flags can be found in Section 5.5.1.

### 6.8.2 Evaluation Goals and Evaluation Metric

Because FlexThread aims at supporting the efficient execution of multi-threaded applications with different optimal core allocations, each benchmark was executed with its optimal core allocation in this evaluation. The optimal core allocation for each benchmark can be found in Section 5.5. We run each benchmark with three configurations on both platforms:

1. The first configuration, which is called the "baseline", uses one thread per core with default PThreads and GOMP libraries on Linux. This configuration gives the best performance that users can get when FlexThread is not used.

2. The second configuration, which is called the "dist-sync", uses one thread per core with FlexThread's distributed synchronizations. The second configuration gives the best performance that users can get with distributed synchronization.

3. The third configuration, or the FlexThread configuration, always uses 256 threads with FlexThread. We chose 256 threads, because it provides less-than-5%-overhead guarantee on both of our NUMA platforms, based our analysis in Section 6.6. This third configuration gives the actual performance that users can expect with FlexThread.

Comparing the performance of the three configurations answers the following three questions:

1. What is the performance benefit of using distributed synchronization? We answer this question by comparing the first and second configurations.

2. What is the performance impact of executing large numbers of threads? We answer this question by comparing the second and third configurations.

3. What is overall performance improvement or slowdown of FlexThread (i.e., combined performance impact of distributed synchronization and large numbers of threads) over current thread libraries? We answer this question by comparing the first and third configurations.

Note that any performance improvement observed in this section is from distributed synchronization and large numbers of threads, instead of better core allocations, because all benchmarks are already executed with their optimal core allocations. The total performance benefit of using optimal core allocation and FlexThread over all-cores allocations is presented in the next Chapter(8.

We run each benchmark with each configuration with five trials. We then computed the average execution time for that benchmark under that configuration. We then report the performance improvement of one configuration over another. More specifically, for any two configurations, $conf1$ and $conf2$, the performance improvement of $conf2$ over $conf1$ is defined as,

$$Perf\_Improv = \frac{Time_{conf1} - Time_{conf2}}{Time_{conf2}} \times 100\%. \tag{6.9}$$

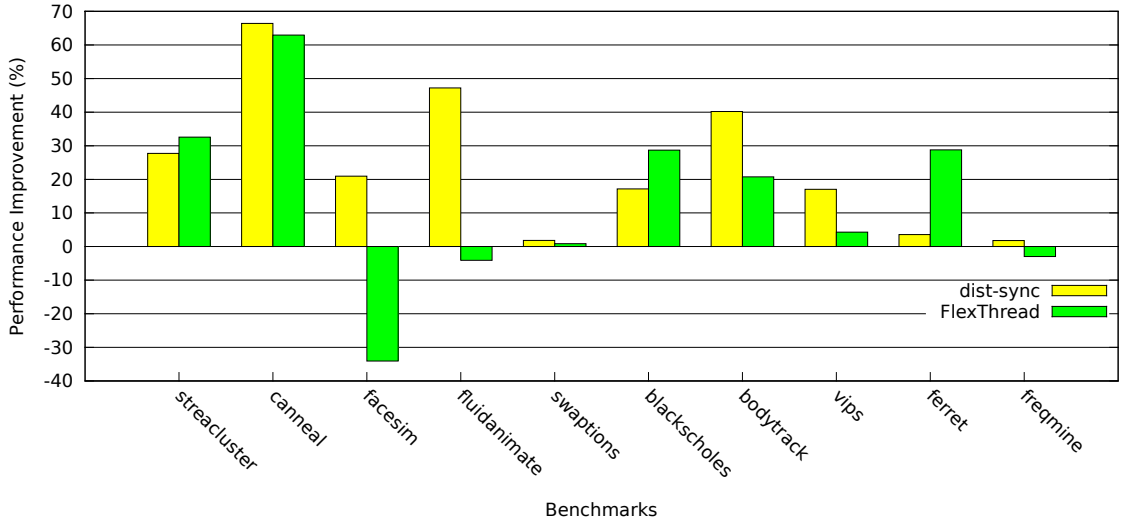
Positive values of $Perf\_Improv$ indicate performance improvement, and negative values indicate slowdown (or overhead).

### 6.8.3 Results for PARSEC Benchmarks

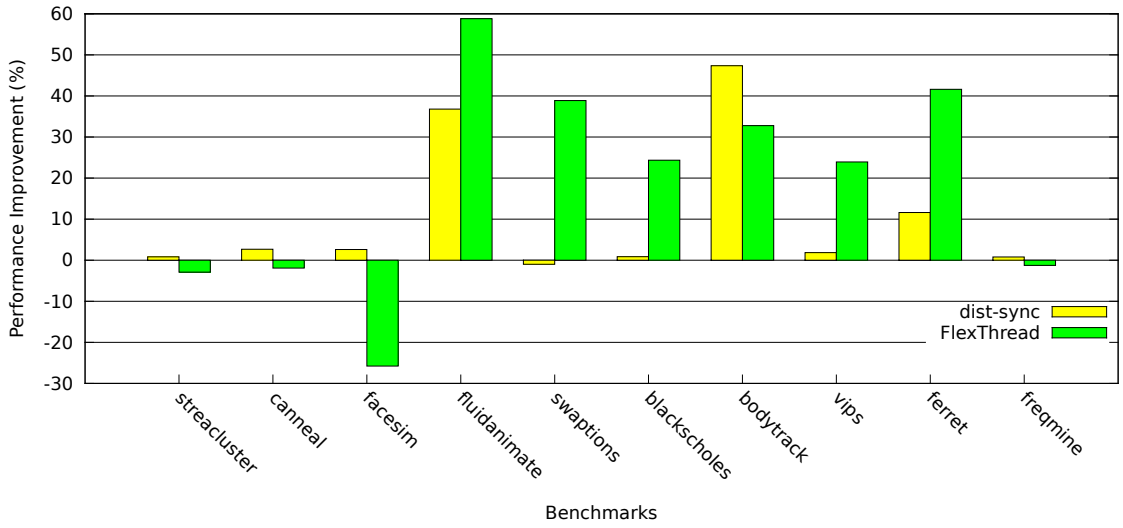
Figure 6.6 gives the performance improvement of the "dist-sync", or the distributed synchronization, and FlexThread configurations over the "baseline" configuration, for PARSEC benchmarks on Intel and AMD NUMA platforms.

#### Performance Improvement of Distributed Synchronization

Figure 6.6 shows that the "dist-sync" performs better than the "baseline" for every benchmark on both machines, suggesting that distributed synchronization improves performance. The average speedup of the "dist-sync", is 17.46% for all PARSEC benchmarks on both platforms.




(a) Results on Intel Platform




(b) Results on AMD Platform

Figure 6.6: Performance improvement of the "dist-sync" and FlexThread configurations over the "baseline" configuration for PARSEC benchmarks on Intel and AMD NUMA platforms. Positive values indicate speedup while negative values indicate slowdown.

### Performance Impact of Large Number of Threads

By comparing the "dist-sync" and FlexThread configurations, we can determine the performance impact of executing large numbers of threads for PARSEC benchmarks.

Executing large numbers of threads, interestingly, does not always mean slowing down. In fact, in Figure 6.6, five benchmarks, *blackscholes*, *ferret*, *vips*, *fluidanimate* and *swaptions*, have performance improvement when executing with 256 threads (FlexThread), than using one thread per core ("dist-sync"). For *blacksholes*, we observed that executing large numbers of threads promoted the data sharing in the L1 cache, thus improving performance. For *ferret*, executing large numbers of threads improved its processor utilization, which in-turn improved performance. *Ferret* has considerable I/O operations, and thus its threads frequently suspend for I/O waits. As a result, *ferret* had low processor utilization when executing with only one thread per core. Similarly, *vips* also enjoyed improved processor utilization and better performance with 256 threads on the AMD platform. For *fluidanimate* and *swaptions* on the AMD platform, their performance improvement came from better load-balancing. Both *fluidanimate* and *swaptions* can only partition their work into smaller jobs with a count of power of 2. However, the AMD platform has 48 cores, and neither *fluidanimate* or *swaptions* can evenly partition their work in 48 small jobs for parallel execution. Hence, the performance of these two benchmarks suffered from unbalanced loads on 48 cores. Whereas, using 256 threads achieves much better load balancing, hence, achieved better performance.

Most of the rest of the benchmarks experienced limited slowdown when executing with large numbers of threads. These benchmarks had less than 5% slowdown with 256 threads. Three cases, which are *bodytrack*, *facesim*, and *fluidanimate* (on the Intel platform only), had slowdown higher than 5%. This slowdown is caused by current Linux scheduler, who frequently switches lock-holding threads out. The threads switched in, however, are quickly switched out because they fail to acquire locks. This scheduling scheme significantly increases context switches and degrades performance. A solution for this problem has been proposed, and we plan to implement this solution in the future [122].

In summary, executing large numbers of threads provides performance improvement in many cases, and the slowdown is usually limited within 5%. If the negative impact of the Linux scheduler is excluded from the results, executing large number of threads provides 9.75% speedup on average, over running one thread per core.

#### Performance Impact of FlexThread

As shown in Figure 6.6, FlexThread provides better performance than current thread libraries ("baseline" configuration) for most PARSEC benchmarks. The highest performance improvement of FlexThread is 62.9%, which is achieved with *cannel* on the Intel platform. The average performance improvement of FlexThread is 16.3% for PARSEC benchmarks on the two platforms. These results suggest that not only does FlexThread support the efficient execution of PARSEC benchmarks with their varying optimal core allocations, it is actually a better option for executing PARSEC benchmarks than current thread libraries, because it provides considerably better performance.

### 6.8.4 Results for NPB Benchmarks

Table 6.2 and Table 6.3 give the performance improvement of the "dist-sync" and FlexThread configurations over the "baseline" configuration, for NPB benchmarks on the Intel and AMD NUMA platforms. Because NPB benchmarks have multiple phases, we present the results for each phase. The function name of each phase is also given in Table 6.2 and Table 6.3.

#### Performance Improvement of Distributed Synchronization

Table 6.2 and Table 6.3 show that the "dist-sync" configuration performs better than the "baseline" configuration for every NPB benchmark on both machines, suggesting that distributed synchronization improves performance. The average speedup of the "dist-sync" configuration is 21.4% for all NPB benchmarks on both platforms.

#### Performance Impact of Large Number of Threads

By comparing the "dist-sync" and FlexThread configurations in Table 6.2 and Table 6.3, we can determine the performance impact of executing large numbers of threads.

Similar to the results of PARSEC benchmarks, executing large numbers of threads also improved performance for several NPB benchmarks, including *mg.D* on the Intel platform, *ep.D* on Intel platform, *is.D* on the AMD platform, *cg.D* on the AMD platforms and *lu.D* on the both platforms. Their performance improved using large

| benchmark | phase | "dist-sync" | FlexThread |
|---|---|---|---|
| mg.D | resid | 5.4% | 15.2% |
| | psinv | 5.0% | 7.8% |
| | rprj3 | 8.8% | 3.7% |
| | interp | 8.5% | 30.9% |
| | norm2 | 0.1& | -0.4% |
| | comm3 | 505.6% | 382.9% |
| sp.D | xsolve | 5.83% | -0.1% |
| | ysolve | 0.84% | -2.4% |
| | zsolve | -2.1% | -4.4% |
| | rhs | 13.7% | 9.0% |
| ep.D | Gaussian Pairs | -1.8% | -1.2% |
| | Random Numbers | 0% | 7.6% |
| cg.D | conjgd | 4.0% | 3.8% |
| bt.D | xsolve | 1.4% | 1.8% |
| | ysolve | 2.0% | 4.2% |
| | zsolve | 0.7% | -2.6% |
| | rhs | 12.4% | 7.8% |
| is.D | benchmarking | -0.3% | -2.9% |
| ft.D | fft | 1.4% | 0.6% |
| | evolve | 17.8% | 17.7% |
| lu.D | rhs | 8.1% | 2.7% |
| | jacld | -3.2% | 22.4% |
| | blts | 23.2% | 23.3% |
| | jacu | 8.0% | 17.0% |
| ua.D | convect | 3.0% | 0.9% |
| average | | 25.1% | 21.8% |

Table 6.2: Performance improvement of the "dist-sync" and FlexThread configurations over the "baseline" configuration for PARSEC benchmarks on Intel NUMA platforms. Positive values indicate speedup while negative values indicate slowdown.

| benchmark | phase | "dist-sync" | FlexThread |
|---|---|---|---|
| mg.D | resid | 0.7% | -4.3% |
| | psinv | 1.6% | -4.9% |
| | rprj3 | 11.8% | 7.1% |
| | interp | 11.4% | 8.1% |
| | norm2 | 6.6& | -2.0% |
| | comm3 | 185.9% | 103.8% |
| sp.D | xsolve | -0.7% | 8.5% |
| | ysolve | -0.5% | 2.0% |
| | zsolve | 7.0% | 7.1% |
| | rhs | 13.7% | 9.0% |
| ep.D | Gaussian Pairs | 2.81% | -1.9% |
| | Random Numbers | 123.3% | 127.1% |
| cg.D | conjgd | 4.2% | 11.2% |
| bt.D | xsolve | 4.8% | 6.3% |
| | ysolve | 3.9% | 2.4% |
| | zsolve | 0.1% | 0.1% |
| | rhs | 9.68% | 5.85% |
| is.D | benchmarking | 0.1% | 32.0% |
| ft.D | fft | -4.3% | 2.8% |
| | evolve | 26.0% | 20.4% |
| lu.D | rhs | 27.5% | 22.4% |
| | jacld | -7.6% | 6.5% |
| | blts | 4.0% | 1.9% |
| | jacu | 7.4% | 15.1% |
| ua.D | convect | 2.9% | -1.9% |
| average | | 17.7% | 15.4% |

Table 6.3: Performance improvement of the "dist-sync" and FlexThread configurations over the "baseline" configuration for PARSEC benchmarks on AMD NUMA platforms. Positive values indicate speedup while negative values indicate slowdown.

number of threads, as fine-grained work partitioning promoted data sharing in the L1 cache, which in-turn reduced the number of local and inter-node memory accesses.

For most NPB benchmarks, the slowdown caused by executing large numbers of threads was less than 5%, suggesting that the overhead of using large numbers of threads is limited. Only in the case of *mg.D* with phase *comm3* did the slowdown become larger than 5%. *Comm3* is a very short phase. Therefore, it has very high synchronization frequency, causing the higher slowdown. However, *comm3* only accounts for less than 1% of the execution time of *mg.D*. As a result, the higher slowdown had little impact on the overall execution time of the benchmark.

#### Performance Impact of FlexThread

As shown in Table 6.2 and Table 6.3, for many NPB benchmarks, FlexThread provides better performance than current thread libraries ("baseline" configuration). The highest performance improvement of FlexThread is 382.9%, which is achieved with *mg.D* on the Intel platform. The average performance improvement of FlexThread is 18.6% for NPB benchmarks on the two platforms. This result suggests that FlexThread supports the efficient execution of NPB benchmarks with their varying optimal core allocations. Moreover, the overall performance benefits of FlexThread suggest that FlexThread should be always used for executing these benchmarks instead of current thread libraries.

## 6.9 Summary

The rigidity of current multi-threaded applications prevents them from being efficiently adapted to their optimal core allocations at run-time. This chapter provides a detailed analysis of this problem and proposes a solution of executing large numbers of threads with distributed synchronization. We implemented this technique into a run-time system called FlexThread. We then evaluated FlexThread on two large-scale NUMA platforms with PARSEC and NPB benchmarks, using their optimal core allocations. The experimental results show that benchmarks managed by FlexThread had less than 5% slowdown compared to directly executing them with one thread per core using their optimal core allocations. More importantly, compared to using one thread per core, FlexThread can also provides up to 382.9% performance improvement. These results suggesting that not only does FlexThread support the

efficient execution of multi-threaded applications with varying optimal core allocations, FlexThread should be used for executing these applications instead of current thread libraries, because of its performance benefits.

# Chapter 7

# REEact: A Customizable Run-time Framework for Multicore Platforms[1]

## 7.1 Introduction

Chapter 4, Chapter 5 and Chapter 6 present models and techniques for the prediction and adaptation of the optimal core allocations for multi-threaded applications running on large-scale NUMA machines. However, these models and techniques cannot directly benefit ordinary users, because applying them requires run-time memory behavior profiling, hardware configuration and application execution management. This chapter describes a run-time framework called REEact, which provides services to support run-time memory behavior profiling, hardware configuration detection, and application execution management.

If we extend our vision to a broader scope, it is easy to see that as the core counts and sophistication of modern chip multiprocessors (CMPs) increase, run-time management techniques are needed to address various resource management challenges. There have been a flurry of such techniques proposed by the research community, such as cache contention management [29, 76, 168], processor temperature management [162, 163], and process variation management [147, 158]. While these and other adaptive policies have shown significant promise, *Users*, such as system administra-

[1]This project was conducted jointly with Dr. Bruce Childers and Dr. Ryan Moore from University of Pittsburgh, as well as Dr. Mary Jane Irwin, Dr. Mahmut Kandemir and Mr. Mahmut Aktasoglu from Pennsylvania State University.

tors, application developers, and other technical experts, need a platform to create, customize, and deploy adaptive resource management policies that address a myriad of design goals and requirements. These policies must also be *plug and play* as user-specific, application-specific and hardware-specific goals change.

For example, consider two users. The users are executing the same application on two identical machines (i.e., the same CMP architecture). However, the users have two distinct requirements. $UserA$ has a tight power budget and prefers a management policy that focuses on minimizing power consumption. $UserB$ desires a management policy that prioritizes performance over power consumption. These goals are contradictory and require distinct management policies. Given the wide range of distinct requirements, it is desirable that management frameworks support flexibility as user requirements, or application requirements, or even hardware characteristics, change.

Currently, most resource management policies are implemented in the operating system (OS) kernel. However, the OS is not well-suited for implementing and incorporating custom policies for two reasons:

1. The OS is not designed to take into account application-specific information when making management decisions. However, with application-specific information, user-level management policies can adjust the execution of applications in a way that is not possible with current OSes. For example, consider an application that uses work-stealing. Work-stealing allows the number of worker threads spawned by the application to be dynamically adjusted. If details of the work-stealing design are known, the management policy can dynamically increase the number of the application's worker threads when the system is underutilized, and reduce the number of threads for better fairness when the system is over-utilized.

2. The complexity of modifying OS policies or adding new ones to the OS is high, which can prevent users from designing their own policies. Even after a custom policy is implemented and carefully tested, much effort has to be made when the same policy is ported to another OS or the user-goal is changed. Furthermore, custom policies in OS kernels may introduce security issues if they are not carefully designed and tested.

Implementing resource management policies at the user-level is easier, and it allows utilization of application-specific information. Previous work has proposed different techniques for user-level resource management [8, 35, 47, 169]. However, these

techniques do not necessarily provide easy customization by the user, and some techniques only target a subset of resource management problems of CMPs. Moreover, some techniques do not provide the means to utilize online performance monitoring technology available in modern CMPs. This monitoring capability is widely used by many policies to monitor the system and dynamically adjust management decisions [29, 76, 147, 158, 162, 163, 168].

In this dissertation, we advocate using a user-level run-time system to provide a framework for easy integration and development of custom CMP resource management policies. To design a user-level run-time framework, there are several challenges to overcome. The first challenge is to provide the necessary resource management facilities to allow easy development of customized resource management policies. These facilities include allocating hardware resources, adjusting application execution, and collecting run-time information about the resource landscape and application status. Furthermore, the run-time-implemented policies should dynamically adapt based on the actual run-time environment. Additionally, the run-time framework should be carefully designed so that management overhead does not outweigh the benefit of a custom management policy.

This chapter presents a *Customizable Virtual Execution Manager* (REEact) which provides the flexibility to implement dynamic custom resource management policies. Situated between applications and the OS and hardware, REEact is active at run time, and can use both application and hardware run-time information to manage and coordinate the applications running in the system.

REEact provides the capability to specify custom management policies that need dynamic adaptation. It offers basic services for resource and application management to permit the incorporation of different management and coordination policies and mechanisms, including those that are customized to a workload, computing environment and/or system goals. These services are exposed through easy-to-use application programming interfaces (API), allowing quick development and testing without the burden or difficulty of modifying global OS policies.

With REEact, custom policies can be easily ported across platforms, as long as a few basic facilities are available on the target platform, such as thread pinning and access to hardware performance counters. Currently, REEact supports two operating systems, Linux and Solaris, and two ISAs, x86 and SPARC. Moreover, although we introduce a new layer into the system, REEact is very lightweight. Its overhead is typically less than 3% (see Section 7.3).

To demonstrate REEact's capabilities and usefulness for run-time management, we present two case studies. The first case study examines how REEact can implement a custom thermal management policy for a malfunctioning machine. The second case study describes using REEact to dynamically control hardware prefetchers to reduce power consumption without sacrificing performance.

The contributions of this chapter include:

1. The REEact framework that provides the capability to easily write user-specific, application-specific and hardware-specific management policies with dynamic adaptation. We describe REEact's software architecture, which is designed to be easy-to-use, extensible, configurable and portable. REEact also permits the implementation of policies that consider application semantics.

2. A thorough evaluation of the overhead and scalability of REEact on a 32-core NUMA platform. We demonstrate that, by careful design and implementation, a user-level virtual execution environment, like REEact, can perform aggressive and fine-grained on-line monitoring, dynamic adaptation and multi-application/thread coordination with very low overhead (<3%) and high scalability (64 thread contexts).

3. Presentation of two case studies that demonstrate REEact's flexibility for providing custom dynamic resource management. Evaluation of the two custom policies show the flexibility, effectiveness and low overhead of REEact. The results also highlight the benefits of customization. Over conventional systems, case study 1 improves performance by up to 16%; and case study 2 improves performance by up to 69%, energy consumption by up to 43%, and energy-delay-product by up to 142%.

This chapter is organized as follows. Section 7.2 presents the high-level structure and operation of REEact. Section 7.3 evaluates the overhead of REEact. Section 7.4 illustrates the operation and utility of REEact by presenting two case studies where REEact manages the use of CMP resources as by specified policies. Section 7.5 concludes this chapter.

## 7.2 REEact Framework

This section provides an overview of the REEact software architecture, and it describes the design of the framework and implementation choices.

### 7.2.1 REEact Software Architecture Overview

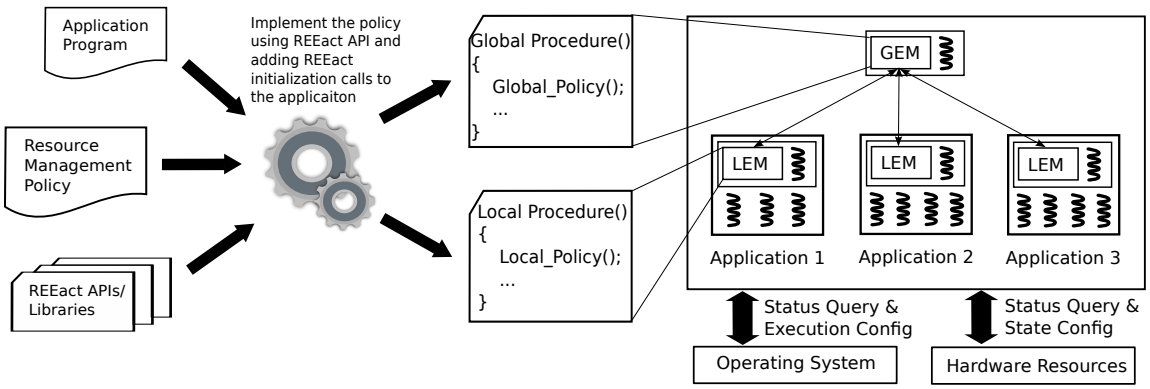


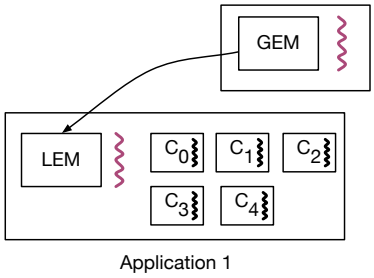

Figure 7.1: A custom policy requires implementing two procedures using the REEact API.

Figure 7.1 sketches the flow of using REEact to implement the specified policies. In REEact, a specified policy requires the implementation of two procedures using the API provided by REEact: a global procedure that manages the execution of multiple applications, and a local procedure that manages the execution of an application (and its threads).

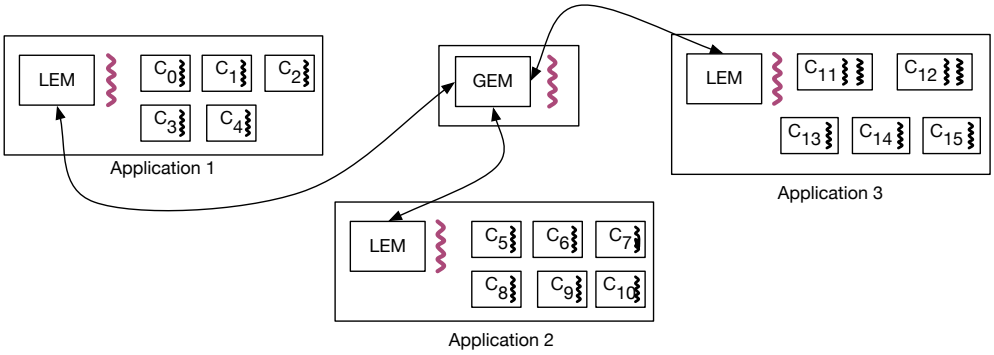
The modification to an existing application to use REEact is straightforward—the main program of the application is modified to include a call to the REEact execution manager. This call essentially places the control of the main thread of execution (as well as subsequent threads the application may create) under the control of REEact. No further modification to the application is required.

During execution, REEact is initialized and it invokes the two procedures of the custom policy, and carries out the specified operations. By coordinating with the OS and the hardware, these operations manage applications and hardware resources.

To illustrate REEact's structure and operation, we first present a simple example. In this example, REEact manages the execution of three multi-threaded applications on a 16-core CMP. For ease of explanation, each core has a single execution context (i.e., simultaneous multi-threading or hyper-threading is not supported). For this




(a) REEact management of a single, multi-threaded application.




(b) REEact management of three, multi-threaded applications.

Figure 7.2: REEact dynamically allocated/deallocated resources (i.e., cores) using a custom FCFS policy.

example, REEact manages the use of the computation resources (i.e., the cores) and dynamically allocates or deallocates them during thread creation and termination following a specified first-come, first-served (FCFS) policy.

In this example, the three applications, $App_1$, $App_2$, $App_3$, create five, six, and seven threads, respectively. We assume that $App_1$ acquires all its resources before $App_2$ begins execution, and $App_2$ acquires all its resources before $App_3$ begins execution.

Initially, REEact initiates a global execution manager (GEM), which invokes the global procedure to manage multiple applications. In this example, all 16 cores are managed by the GEM. Since GEM's execution has very low overhead, it is allowed to execute on any core (even a core that is allocated to an application thread). When $App_1$ begins execution, REEact is loaded by the OS, and REEact invokes its initialization API routine. This call creates a local execution manager (LEM), which invokes the local procedure to manage the application. The first action of the LEM

is to communicate with the GEM and request a core. From this point on, the LEM executes on any core that has been allocated to it (at this point it has one core).

As $App_1$ executes, it creates other threads. REEact intercepts thread creation calls, and notifies the LEM. The LEM communicates to the GEM requesting another core (recall the policy is FCFS). At this point, the GEM has available cores and one is allocated to this LEM. As application execution continues, additional threads are created and cores are allocated in a similar manner.

Figure 7.2a illustrates the structure of REEact at this point. In the figure, five cores ($C_0$–$C_4$) have been allocated to $App_1$—one for each thread of the application. The LEM thread, like the GEM thread, has low overhead and is permitted to run on any of the cores allocated to the application (i.e., $C_0$–$C_4$).

When $App_2$ starts, the same process occurs. Here, $App_2$ creates six threads and is therefore allocated six cores ($C_5$–$C_{10}$). Then $App_3$ begins execution and requests cores (there are now five unallocated cores remaining). The first five thread creations result in a core being allocated for each thread. However, the sixth thread creation results in the GEM informing $App_3$'s LEM that no core is available. The LEM then must map this thread to a core already allocated to it. The process is similar for the seventh and final thread. The final state of execution is illustrated in Figure 7.2b. The last two threads of $App_3$ are mapped to cores $C_{11}$ and $C_{12}$, respectively.

Note that during the execution of an application, a thread may terminate. REEact intercepts thread termination and notifies the applications' LEMs. If the termination of a thread frees a core, the core may be dynamically reallocated to other threads. The LEM may map an existing thread to the core, or return the core to the GEM for global reallocation. In this example, if a thread in $App_1$ terminates, the core is returned to the GEM which makes a global decision to offer it to $App_3$. $App_3$ can then map one of the threads that is sharing a core to its own core.

REEact supports managing applications with multiple policies. Currently, individual policies are combined manually. REEact also permits the co-existence of multiple GEMs, where each GEM controls some applications and resources using different policies.

| API Component | API Methods | Notes |
|---|---|---|
| Communicator | sendMessage | |
| | getMessage | blocking |
| | timeoutGetMessage | non-blocking with timeout |
| Monitor (HW Status) | readPMUperThread | |
| | readPMUperCore | |
| | getCoreTemperature | |
| | getTemperatureThreshold1 | threshold temperature of DVFS |
| | getTemperatureThreshold2 | threshold temperature of core shutdown |
| | getCoreFrequency | |
| | getHWComponentState | e.g. whether prefetcher or L2 cache is disabled? |
| Monitor (Sys. Util.) | getCoreUtilization | |
| | getSystemLoad | |
| SW Actuator | pinAppsToCores | |
| | pinThreadsToCores | |
| HW Actuator | enableHWComponent | e.g. enable or disable hardware prefetchers |
| | adjustFrequency | adjust processor/core frequency or duty cycle |
| App. State Tbl. (GEM) | getUnallocCores | |
| | getAllocCores | |
| | allocOrDeallocCore | |
| | getTotalCoreCount | get the total number of cores |
| | getCoresofCache | get the cores that share a cache |
| | getAllL2Caches | get a list of available L2 caches |
| App. State Tbl. (LEM) | getCurrentlyUsingCores | get the cores allocated to this LEM |
| | getAppThreads | |

Table 7.1: API component and their associated methods currently provided by REEact

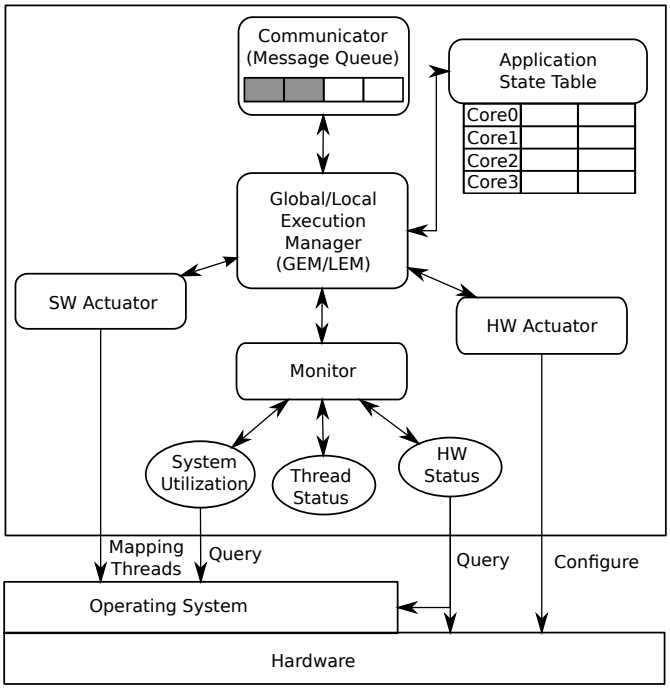



Figure 7.3: Essential components of REEact.

### 7.2.2 REEact Design

#### REEact Components

REEact provides the framework and services to enable the implementation of management policies (essentially the global and local procedure). These services, such as GEM/LEM communication and thread mapping, are provided through various REEact components.

Figure 7.3 shows the essential components of REEact. Because proper resource management decisions have to be made based on the actual states of applications and hardware resources, REEact has monitors to collect their run-time status. Hardware and software actuators are provided to adjust resource allocation and application execution. A GEM makes global management decisions based on the run-time information collected by the LEMs, and the communication component provides facilities for communication between GEMs and LEMs. REEact also provides application status tables so that GEM/LEM can keep track of resource allocation. The following paragraphs briefly describe each component.

The **Global/Local Execution Managers (GEM/LEMs)** have three major duties. First, they initialize and release REEact component objects during application start-up and termination. Second, they maintain the tree structure introduced in Section 7.2.1 (Figure 7.2b). Third, they execute customized policy procedures.

The GEM and LEMs are instantiated during application start-up. When several applications execute simultaneously, the application that starts first becomes the "master" which creates the GEM and the first LEM. Applications that start later only create LEMs. The GEM and LEM perform all necessary operations to create other REEact components. A LEM also identifies the GEM and creates a two-way communication link between them. And lastly, the GEM/LEM invokes the specified policy procedures (see Figure 7.1).

The **Communicator** transfers data among the GEM and LEMs asynchronously. It is designed as a message queue attached to a GEM/LEM. Each message is composed of three parts: a sender ID, a message type and the message body. The actual meaning of the message is policy dependent.

The **Monitor** provides capabilities to monitor the status of both the hardware and the applications. It collects information about the hardware, system utilization, and the status of threads.

Hardware information that can be collected includes the output of performance

monitoring units (PMU), core frequency, core temperature, and whether a particular hardware component (e.g., prefetchers, L2 caches, etc.) is enabled.

The system utilization information includes the load (utilization) of each core, and the overall load. REEact also monitors the status of application threads, such as thread creation, termination and suspension.

The **SW Actuator** configures the execution of applications. The SW actuators map both applications and threads to cores. How cores are allocated depends on the policy used.

The **HW Actuator** configures a hardware (HW) component. A HW actuator can enable or disable a hardware component of a core (e.g., prefetchers), or adjust processor frequency by setting special bits of model specific registers (MSR).

The **Application Status Table (AST)** contains the current states of hardware resources and applications. Each GEM and LEM has its own AST, which contains the information about the resources and threads that it controls. ASTs can be extended to include policy-required information.

### REEact API

REEact components are represented as objects and their services are exposed through methods that operate on these objects. The core of REEact is the GEM and LEM classes and associated methods.

Table 7.1 lists the methods currently provided. For communication among the multiple applications, we provide methods to send and receive messages as part of the communicator component. There are two methods for receiving messages—a blocking method and method with timeout. Currently, the methods for monitors and actuators are designed to provide access to typical resource controls [78, 82, 85, 145, 163, 166, 168]. We provide the methods for the monitor component to collect the profiling information either per-core or per-application-thread basis. There are additional methods to obtain current temperature, temperature threshold, frequency and other hardware status of the individual physical cores. We also provide methods to gather information about core utilization. The methods for the software actuator component enable the control of the application using software, including on which core an application's threads run. The methods for a hardware actuator enable the control of the state of a particular hardware device, for example, to enable or disable the hardware prefetchers. The methods for the application state table enable the gathering of information about all the applications running in the system for the

GEM and one application for LEMs. Additional services and features will be added as REEact expands to support more operating systems and architectures.

### 7.2.3 REEact Implementation

The next paragraphs describe the actual implementation of REEact components on Linux-x86 and Solaris-SPARC.

**Global/Local Execution Manager (GEM/LEM)**: A GEM/LEM is implemented as a helper thread, which is created during REEact initialization at the beginning of application execution (recall that a REEact initialization call is automatically invoked by the OS when REEact library is attached to a new process).

**Communicator**: The communicator is implemented using shared memory and semaphores. The message queue of a GEM or LEM is essentially a portion of memory that is shared by all GEM/LEMs. Therefore, posting a message or reading a message is an access to this shared memory. Each queue is associated with a semaphore, which notifies the GEM/LEM on arrival of a message.

**Monitor**: On Linux-x86, the monitor uses Perfmon2 to read PMU data [48]. On Solaris-SPARC, the monitor uses "Libcpc" to read PMU data.

On x86 architectures, the core frequency, temperature and other hardware component states are acquired by reading MSRs. REEact reads the MSRs through a special driver that we designed and implemented. On SPARC, these hardware states are acquired from the OS.

System utilization information (e.g., processor utilization) is acquired from the OS. For example on Linux, this information can be read through the "stat" file (per-core) or "loadavg" file (all-cores) under "/proc". On Solaris-SPARC, this information can be acquired from library "Libkstat".

To intercept thread status changes (e.g., thread creation, termination, suspension), we dynamically link application thread functions (e.g., pthread_create, pthread_join) to REEact's thread functions using "LD_PRELOAD". Once a thread status change is detected, the monitor notifies the GEM and LEM by sending messages to them.

**SW Actuator**: For thread mapping, the SW actuator uses the core-affinity system call. Core/processor affinity is readily available in today's commodity operating systems, including Linux, Solaris and Windows. Once a thread is mapped to a set of cores, the OS does not move the thread to use any other cores. Less rigid mechanisms could also be used; for example, a mechanism could be provided to convey schedul-

| Workload size | Benchmark Initials used in each workload |
|---|---|
| $\|WL\|$=1 | {BS}, {BT}, {CN}, {FA} |
| $\|WL\|$=2 | {BS, BT}, {BS, CN}, {BS, FA}, {BS, SW} |
| $\|WL\|$=4 | {BS, BT, CN, FA}, {BS, CN, FA, SW}, {BS, BT, CN, FA} |
| $\|WL\|$=8 | {BS, BT, CN, FA, FE, FL, SC, SW} |
| $\|WL\|$=16 | {BS, BT, CN, FA, FE, FL, SC, SW, BS, BT, CN, FA, FE, FL, SC, SW} |

Table 7.2: The benchmarks used in each workload for measuring REEact overhead. $|WL|$ denotes the number of applications in a workload, e.g., $|WL| = 2$ means two applications in a workload. For each workload size, each set in the next column represents one workload composed by $|WL|$ number of PARSEC benchmarks (indicated by the acronym).

ing and allocation hints about thread co-location to the operating system. These mechanisms could be targeted by a user policy in REEact to guide OS management decisions.

**HW Actuator**: The HW actuators enable/disable hardware components, and adjust core frequency by setting special bits of MSRs. We implement a driver that allows reading and writing MSRs at the user-level.

**Application Status Table (AST)**: In the default implementation, the ASTs are lists that store resource allocation information. Policies can define their own lists (or other data structures) to store any data they need.

## 7.3 REEact Overhead Evaluation

This section evaluates the overhead of the REEact framework implementation. REEact incurs overhead when reading hardware performance counters, by monitoring and managing thread creation, and through communication between the LEMs and GEM.

To determine the run-time overhead of REEact, we chose several multithreaded applications from PARSEC [17], and we measured REEact's overhead as we scaled the total number of threads and applications. The experiments were run on a CMP machine that has four Intel Xeon X7550 processors each of which has eight cores. As the processors are hyper-threaded, each core has two thread contexts and consequently the machine supports 64 thread contexts. Each physical core has a private 32KB L1-cache, a 256KB L2-cache and one 16MB L3-cache shared by eight physical cores. This machine is running Linux 2.6.32.

We conducted a series of experiments with workloads consisting of different num-

| | Number of threads | | | |
|---|---|---|---|---|
| | 1 | 2 | 4 | 8 |
| $\|WL\|$=1 | 0.77 | 1.36 | 1.46 | 2.74 |
| $\|WL\|$=2 | 0.79 | 1.82 | 1.45 | 0.63 |
| $\|WL\|$=4 | 0.6 | 0.84 | 0.94 | 2.49 |
| $\|WL\|$=8 | 1.43 | 1.07 | 1.18 | 1.37 |
| $\|WL\|$=16 | 1.27 | 1.47 | 1.32 | — |

Table 7.3: Total overhead (in percentage) as the number of applications and threads per application is varied, for counter-reading period=10 milliseconds. $|WL|$ denotes the number of applications in a workload, e.g., $|WL| = 2$ means two applications in a workload.

bers of randomly chosen PARSEC benchmarks. The benchmarks used in the experiments are: *blackscholes (BS)*, *bodytrack (BT)*, *canneal (CN)*, *dedup (DD)*, *facesim (FA)*, *ferret (FE)*, *fluidanimate (FL)*, *streamcluster (SC)* and *swaptions (SW)*. To measure the overhead of reading performance counters, we varied the period of reading counters from 10 milliseconds to 32 seconds for a fixed number of applications and threads per application. As we increased the period, REEact's overhead decreased because of less frequent access to the performance counters.

To measure REEact's overhead, we varied the number of applications from 1 to 16 and threads per application from 1 to 8, for a fixed counter-reading period. We set the counter-reading period to 10 milliseconds. We experimentally checked PARSEC and SPEC, and discovered that no benchmark has a phase shorter than 10ms (similar results are also reported by previous research [43]). Therefore, 10ms is small enough in practice to capture phase changes in the application threads. If an application does have phases less than 10ms, then phase changes may go undetected, and there could be some penalty. However, as the phases are short ($<$ 10ms), any penalty should not be high/significant. We ran experiments with four one-application, four two-application, three four-application, one eight-application and one sixteen-application workload with different number of threads under REEact control. The benchmarks used in the workloads are described in Table 7.2. For each workload, the overhead was measured by calculating the average percentage difference in each application's execution time normalized with respect to the native execution (i.e., no REEact).

Table 7.3 shows the overhead results. We did not measure the overhead for 16 applications with 8 threads per application as the total number of threads exceeds the number of thread contexts for the experimental machine. From the table, we observe

that for a particular workload size, as the number of threads increases (reading across a column), the overhead slightly increases. The increase is due to the increased number of messages. For a fixed number of threads, as we increase the number of applications, the overhead varies slightly. This slight variation is caused by the differences in the workloads. For all 19 experiments, the average overhead of the REEact framework is at most 3%, which is acceptably small. Note that, the overall overhead of REEact given in Table 7.3 includes the overhead of sending messages across processors. The low overall overhead (less than 3%) implies that the use of cross-processor messaging has low overhead and is not a bottleneck for the machines we examined.

## 7.4 Case Studies

This section describes two case study policies that are implemented in REEact to demonstrate its flexibility and usefulness. These policies tackle two different problems: thermal management and performance/energy-consumption management.

### 7.4.1 Case Study 1: Fighting the Broken Screw

In the first case study, we demonstrate how to use REEact to implement a policy that reacts to thermal emergencies. We have a CMP computer that frequently experiences overheating. This computer has an Intel Q6600 quad-core processor. The processor has four cores. Each core has 32KB L1 I-cache and 32 KB L1 D-cache. Every two cores of this processor shares one 4MB L2 cache. The machine is running Linux 2.6.25.

In this machine, one of the four screws that fasten the heat sink and fan to the chip is broken, causing some cores (especially *core0*) to easily reach a very high temperature. Most CMP processors have hardware mechanisms to prevent overheating. For this processor, it uses dynamic voltage/frequency scaling (DVFS) and voluntary core shutdown. By reading the on-die digital thermal sensors (DTS), this processor monitors its cores' temperatures. If any core's temperature exceeds a factory predefined threshold $T_1$, the core's frequency and/or voltage are reduced. If a core's temperature keeps climbing and reaches the critical temperature $T_2$, the core is shutdown temporarily. Although these mechanisms prevent catastrophic overheating, they have negative impact on system performance. If we can detect cores that are vulnerable to

**Policy 1.a** Fighting the broken screw: GEM policy procedure (boldfaced functions are REEact methods)

```
1: /* input parameters: ref to all LEM objs, ref to GEM obj, ref to monitor and ref
   to GEM's app state table */
2: INPUT: List lemList, Gem gem, Monitor m, AppStatTbl ast
3: List freeCores ← ast.getUnallocCores();
4: Int T1 ← m.getTempThreshold1();
5: Int T2 ← m.getTempThreshold2();
6: LOOP
7:    Message msg ← gem.getMessage();
8:    Lem lem ← msg.sender;
9:    Core oldCore ← msg.value;
10:   Core newCore ← oldCore;
11:   IF msg.type = TemperatureAboveT1 THEN
12:      /* search for a core that has a temperature below T1*/
13:      FOR ALL c IN freeCores DO
14:         Int t ← m.getCoreTemperature(c);
15:         IF t < T1 THEN
16:            newCore← c;
17:            BREAK;
18:         END IF
19:      END FOR
20:   ELSE IF msg.type = TemperatureAboveT2 THEN
21:      Int lowestT ← T2;
22:      /* List of cores that "lem" has used and had thermal issue (core temperature
         above T2) */
23:      List hotCores ← lem.ast.getCoresAboveT2();
24:      /* search for an unallocated core that has lowest temperature and no thermal
         issue for "lem" yet */
25:      FOR ALL c IN freeCores DO
26:         t ← m.getCoreTemperature(c);
27:         IF  hotCores.doNotHave(c) AND (t < lowestT) THEN
28:            newCore ← c;
29:            lowestT ← t;
30:         END IF
31:      END FOR
32:   END IF
33:   /* ask LEM "lem" to run on newCore */
34:   IF newCore ≠ oldCore THEN
35:      lem.sendMessage(runOnCore, newCore);
36:      freeCores.add(oldCore);
37:      freeCores.remove(newCore);
38:   END IF
39: END LOOP
```

**Policy 1.b** Fighting the broken screw: LEM policy procedure (boldfaced functions are REEact methods)

```
1: /* input parameters: ref to this LEM obj, ref to GEM obj, ref to monitor and
   ref to this LEM's app state table */
2: INPUT: Lem lem, Gem gem, Monitor m, AppStatTbl ast
3: Int T1 ← m.getTempThreshold1();
4: Int T2 ← m.getTempThreshold2();
5: /* extend ast with a new list for the cores on which this LEM has thermal issue
   (core temperature above T2) */
6: List ast.CoresAboveT2 ← new List();
7: LOOP
8:    /* single-threaded app has only one thread and one core*/
9:    Core curCore ← ast.getCurrentlyUsingCores();
10:   Thread appThread ← ast.getAppThreads();
11:   Int coreT ← m.getCoreTemperature(curCore);
12:   IF coreT ≥ T2  THEN
13:      ast.CoresAboveT2.add(curCore);
14:      gem.sendMessage(TemperatureAboveT2, curCore);
15:   ELSE IF coreT ≥ T1 THEN
16:      gem.sendMessage(TemperatureAboveT1, curCore);
17:   END IF
18:   /* Get message with timeout "timeout" (non-blocking)*/
19:   Message msg ← LEM.timeoutGetMessage(timeout);
20:   IF  (msg ≠ NULL) AND (msg.type = runOnCore) THEN
21:      lem.pinThreadstoCores(appThread, msg.value);
22:      sleep(timeout);
23:   END IF
24: END LOOP
```

overheating, and schedule the threads to use these cores less frequently, then we can achieve better performance as well as prevent unnecessary overheating of the cores.

REEact's monitoring methods support reading core temperatures (using MSR "IA32_THERM_STATUS"), as well as dynamically detecting threshold temperatures $T_1$ and $T_2$. To solve the broken-screw thermal problem, we developed a custom policy using REEact. In this policy, the GEM and LEMs work together to detect overheating and migrate application threads appropriately. Policy 1.a and 1.b give the pseudocode for the global (GEM) and local (LEM) procedures respectively. When an application first requests a core for its newly created thread (through its LEM), the GEM randomly allocates a free core to it. During execution, the LEM periodically (every 10 seconds) checks if any of its cores are overheating. Depending on the level of overheating, the LEM requests different cores:

1. If the overheating core's temperature is between $T_1$ and $T_2$, the LEM asks the GEM for a core with temperature below $T_1$ (Policy 1.b line 16). If the GEM finds such a core (Policy 1.a lines 11-19), it responds with the new core (Policy 1.a lines 35-37), to which LEM maps its threads (Policy 1.b line 21).

2. If the overheating core's temperature is equal to or higher than $T_2$, the LEM marks this core as a hot core, and asks the GEM for the coolest non-marked core (Policy 1.b line 13-14). Once the GEM responses with a new core (Policy 1.a lines 21-31), the LEM re-maps its threads to it (Policy 1.b line 21). If the GEM fails to find a core, the LEM and its threads stay on the hot core.

We evaluated this policy using the SPEC2006 benchmarks. Figure 7.4 shows the performance (execution time) improvement of SPEC benchmarks controlled by REEact with the overheating prevention policy, as well as manually tuned optimal execution, compared to the Linux default scheduler, which does not consider core temperature and can freely use any core. The optimal execution time is determined by trying all possible thread-to-core mappings, and choosing the one with the minimum execution time.

The maximum speedup of REEact over the Linux default scheduler is 16%, and the average speedup is 9.5%. The results show the benefit of using REEact framework by implementing this adaptive policy customized for this specific machine. The results also show that REEact and this policy has very low overhead, with at most 3% slowdown compared to optimal results. This slowdown is partially caused by the reactive nature of the policy: it only migrates threads when overheating actually occurs.

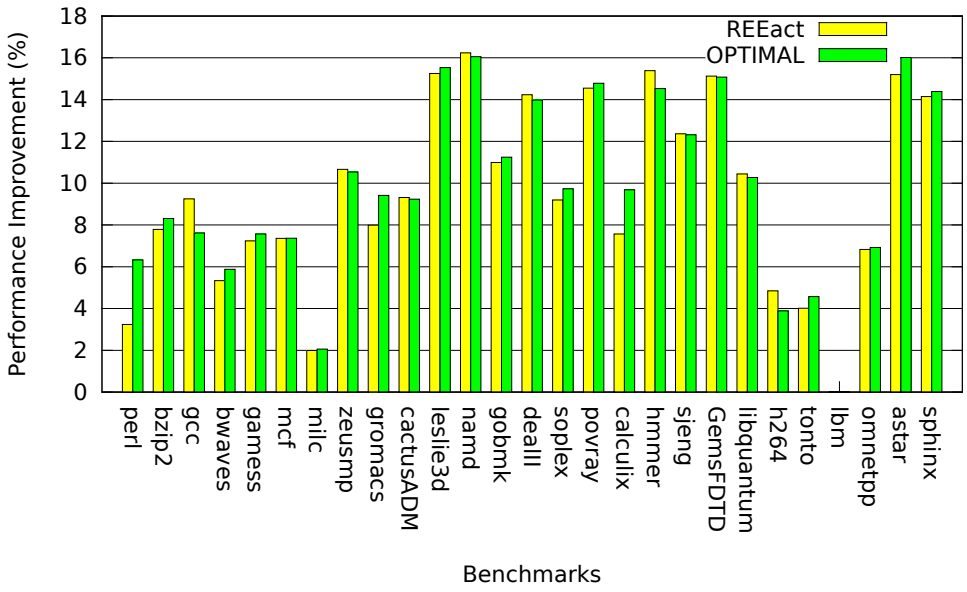



Figure 7.4: Performance (execution time) improvement of SPEC with REEact broken-screw policy and manually tuned optimal execution, compared to the Linux default scheduler.

The policy can be further refined to be proactive instead of being reactive [163].

### 7.4.2 Case Study 2: Reducing Energy Usage without Performance Penalty

In the third case study, we present a REEact policy for reducing system energy consumption. This policy is based on the following observations: hardware prefetchers may have no or negative performance impact if many cache lines prefetched are not used by the application [139]. However, fetching these useless cache lines requires extra energy. Therefore, we can disable the prefetchers when they do not improve performance to save energy. With REEact, we can dynamically examine how the applications use prefetchers, and control the prefetchers accordingly.

Policy 2.a and 2.b give the pseudo-code for the global (GEM) and local (LEM) procedures. When a new thread is created or unblocked (from synchronization wait), REEact's thread status monitor sends a new message to the GEM (Policy 2.a line 6). Upon receiving this message, the GEM starts to test two configurations: one with prefetchers enabled and one with prefetchers disabled. The GEM also notifies the LEMs to collect the number of instructions retired for these two configurations

**Policy 2.a** REEact Power management: GEM policy procedure (boldfaced functions are REEact methods)

```
1: /* input parameters: ref to all LEM objs, ref to GEM obj, ref to monitor, ref to
   GEM's app state table, ref to HW actuators */
2: INPUT: List lemList, Gem gem, Monitor m, AppStatTbl ast, HWActuator hwAct

3: Int coreCnt ← m.getTotalCoreCount();
4: LOOP
5:   Message msg ← gem.getMessage();
6:   IF msg.type = ThreadActivated THEN
7:     /* start sampling configuration with prefetchers on*/
8:     /* enable prefetchers on all cores */
9:     hwAct.enableHWComponent(c, prefetcher, on);
10:    FOR ALL lem IN lemList DO
11:      lem.sendMessage(readInsnRetired, NULL);
12:    END FOR
13:    Int insnPfOn ← 0;
14:    FOR ALL lem IN lemList DO
15:      Message msg ← gem.getMessage();
16:      insnPfOn ← insnPfOn + msg.value;
17:    END FOR
18:    /* start sampling configuration with prefetchers off*/
19:    hwAct.enableHWComponent(c, prefetcher, off);
20:    FOR ALL lem IN lemList DO
21:      lem.sendMessage(readInsnRetired, NULL);
22:    END FOR
23:    Int insnPfOff ← 0;
24:    FOR ALL lem IN lemList DO
25:      Message msg ← gem.getMessage();
26:      insnPfOff ← insnPfOff + msg.value;
27:    END FOR
28:    /* Comparing two configuration */
29:    BOOL switch ← off;
30:    IF insnPfOn > insnPfOff × 1.02 THEN
31:      switch ← on;
32:    END IF
33:    hwAct.enableHWComponent(c, prefetcher, switch);
34:  END IF
35: END LOOP
```

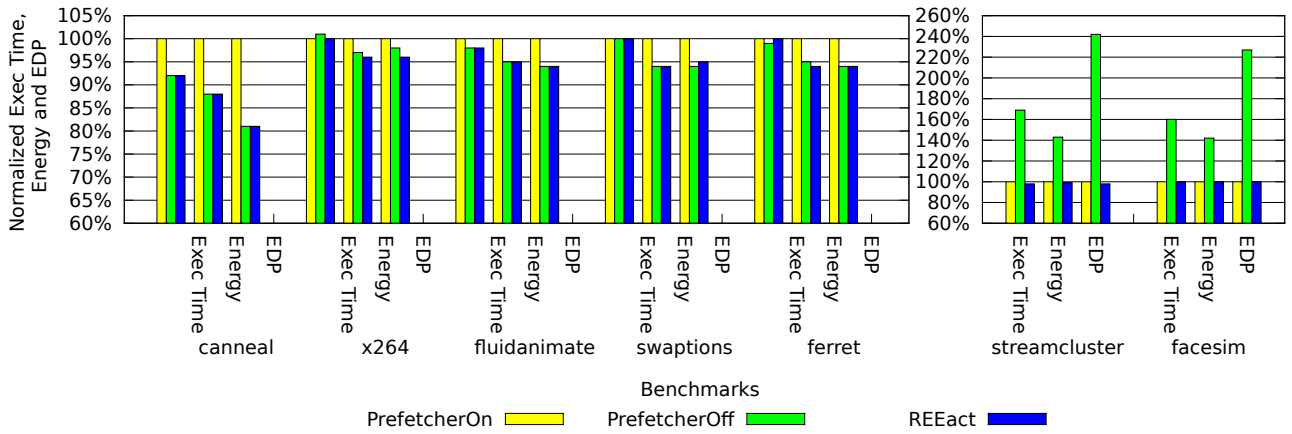



Figure 7.5: Execution time, energy consumption and EDP for PARSEC benchmarks running under REEact and two static configurations where prefetchers are always enabled or disabled (normalized to the configuration where prefetchers are always enabled; the lower bar is better).

(Policy 2.a lines 7-27). Each configuration is executed for 0.5 seconds and each LEM reports the number of instructions retired to GEM (Policy 2.b lines 8-13). The GEM compares the results, and disables the prefetchers if the configuration with enabled prefetchers does not retire more instructions (Policy 2.a lines 28-33). To avoid measurement errors from PMUs, the configuration with enabled prefetchers is considered superior only if it has at least 2% more instructions retired.

We evaluated this policy on a computer with an Intel quad-core processor Q9550. Each core has 32KB L1 I-cache and 32KB L1 D-cache. There are two 6MB L2 caches each shared by two cores. For each L1 cache, there are two prefetchers (DCU and IP) that prefetch data into it. For each L2 cache, there are also two prefetchers (hardware prefetcher and adjacent cache line prefetcher) that prefetch data into it. The DCU prefetcher and the adjacent cache line prefetcher prefetch the next cache line of the missed cache line into the cache. The IP prefetcher and the hardware prefetcher look for a stride in the memory access pattern and prefetch the next expected data [67]. We tested eleven PARSEC benchmarks individually. Each benchmark was configured to run with four threads using native input sets. We ran each benchmark for three iterations and computed the average energy consumption and execution time of PARSEC's "region of interest" (parallel region). We collected the energy consumption of the processor using the methodology from Esmaeilzadeh et al. [50].

Figure 7.5 shows the results of seven benchmarks running under REEact, as well as two other static configurations where prefetchers are always enabled or disabled.

Four benchmarks, *bodytrack*, *raytrace*, *vips* and *blackscholes*, are omitted because configuring prefetchers does not impact their performance or energy consumption. The results show that REEact always provides the best performance and lowest energy consumption. Compared to always enabling prefetchers, REEact improves execution time by up to 8%. It improves energy consumption by 12% and the energy-delay-product (EDP) by up to 19%. Compared to always disabling prefetchers, REEact improves execution time by up to 69%, energy consumption by 43% and EDP by up to 142%. The results show that REEact has little (at most 1%) overhead over the best static prefetcher configuration.

**Policy 2.b** REEact power management: LEM policy procedure (boldfaced functions are REEact methods)

```
1: /* input parameters: ref to this LEM obj, ref to GEM obj, ref to monitor and
   ref to this LEM's app state table */
2: INPUT: Lem lem, Gem gem, Monitor m, AppStatTbl ast
3: List appThreads ← ast.getAppThreads();
4: LOOP
5:    Message msg ← lem.getMessage();
6:    Double time = 0.5; /* sample for 0.5 seconds */
7:    IF msg.type = readInsnRetired THEN
8:       /* read InsnRetired for all the threads of this LEM */
9:       List insnCnts ← insnCnt +
         m.readPMUperThread(appThreads, InsnRetired, time);
10:      gem.sendMessage(InsnRetiredValue, insnCnts.sum());
11:   END IF
12: END LOOP
```

## 7.5 Summary

Various user requirements, application behaviors, and hardware configurations, have greatly complicated the management of CMP hardware resources and applications. Management policies addressing such variations depends on user/application/hardware-specific requirements and require dynamic adaptation. This chapter tackles these management problems by enabling the design and implementation of custom resource management policies with REEact, a user-level run-time framework. This framework provides several services for dynamic management and coordination of hardware resources and applications. REEact allows easy development of custom policies to meet

different user requirements. It also facilitates the development of policies that consider application-specific information and hardware characteristics. We describe the design and implementation of REEact, and use it with two case studies, each focusing on distinct issues on CMPs—thermal management, performance,and power management. These case studies highlight the benefits brought by REEact, such as flexibility, effectiveness, ease-of-use, and separation of concerns. Through these case studies, we also demonstrate that a carefully designed user-level run-time system, like REEact, can effectively manage resources and applications with little run-time overhead.

For the problem of processor over-provisioning, REEact framework provides valuable run-time support of memory behavior profiling, hardware configuration detection and application execution management, allowing our optimal core allocation prediction model to be applied to real life practice.

# Chapter 8

# OptiCore: Automatic Optimal Core Allocation for Multi-threaded Applications on Large-scale NUMA Platforms

## 8.1 Introduction

Through Chapter 4 to Chapter 6, we have presented models and techniques which enable the run-time prediction of, and the dynamic adaptation to, optimal core allocations for multi-threaded applications. In this chapter, we present the comprehensive run-time system, OptiCore, which combines these models and techniques to automatically execute multi-threaded applications with their optimal core allocations.

When executing a multi-threaded application, a user only has to take two extra steps to put this application under the management of OptiCore. First, the user needs to set the value of the environment variable "LD_PRELOAD" to be the path of the OptiCore library. When a multi-threaded application starts, the Operating System queries the value of "LD_PRELOAD", and automatically loads the OptiCore library, which then takes over the control of the execution of this application. Additionally, a user should instruct the application to create a large number of threads. Nearly all multi-threaded applications allow user-specified thread counts, simply by setting command line arguments or environment variables. The exact thread count to use can be determined using the process described in Section 6.6. After these two steps, users

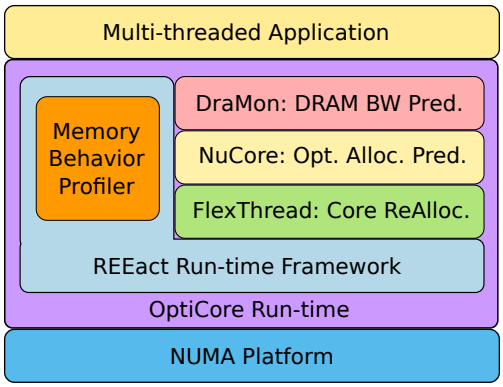



Figure 8.1: The system architecture and components of OptiCore.

can invoke their applications in the same way as they normally do when OptiCore is not involved. After an application starts, OptiCore automatically takes control of its execution, and adapts it to its optimal core allocation. The whole process of predicting the optimal core allocation, and adapting the application to this core allocation, requires absolutely no involvement from the user.

We evaluated OptiCore on two real large-scale NUMA platforms with benchmarks from PARSEC and NPB benchmark suites [15, 75]. The experiment results show that OptiCore performs 34.6% faster than the use-all-cores allocation on average. More importantly, the speedup of OptiCore is achieved with fewer cores allocated. The minimal optimal core allocation uses only 12.5% of all cores and performs 223.4% better. On average, OptiCore uses only 88.7% of all cores and achieves 34.6% performance improvement. The experimental results also show that OptiCore has only 1.85% overhead compared to manually executing each benchmark with its optimal core allocation.

This chapter presents the design and experimental evaluation of OptiCore. The rest of this chapter is organized as follows. Section 8.2 describes the design of OptiCore. Section 8.3 gives the results of OptiCore's experimental evaluation. Section 8.4 summarizes this chapter.

## 8.2 The OptiCore Run-time system

This section presents the design of OptiCore in detail.

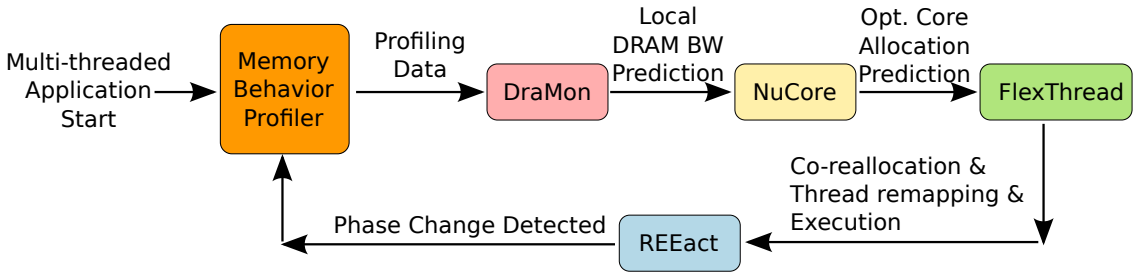



Figure 8.2: The work flow of OptiCore.

### 8.2.1 The System Architecture of OptiCore

Figure 8.1 gives the system architecture of OptiCore. OptiCore implements DraMon (Chapter 4) and NuCore (Chapter 5) models to predict the optimal core allocations for multi-threaded applications. It also incorporates FlexThread (Chapter 6) to efficiently adapt multi-threaded applications to their optimal core allocations during execution. OptiCore is built upon the REEact run-time framework, which is described in Chapter 7. REEact provides services for online application memory behavior profiling, which is required by DraMon and NuCore models. REEact also provides services for thread management, such as thread creation/termination detection, thread migration and phase change detection, which are required by FlexThread and OptiCore to control the execution of multi-threaded applications.

Figure 8.2 gives the flow of OptiCore when it is used to managed a multi-threaded application. After a multi-threaded application starts, OptiCore profiles its memory behavior. Then the profiling data is sent to DraMon for the prediction of the DRAM contention and local DRAM memory bandwidth usages. DraMon's prediction is then sent to NuCore model to predict the optimal core allocations. Once the optimal core allocation is determined, FlexThread remaps application threads to this core allocation to execute. During the execution, REEact continuously monitors phase changes. Once a phase change is detected, the application is profiled again and adapted to use a new optimal core allocation. The whole process requires no user intervention. Moreover, OptiCore does not require any knowledge of, or modification to, the application source code.

### 8.2.2 Handling Phase Changes

Because phase changes in applications can change their memory behaviors, which in-turn change their optimal core allocations, OptiCore has to constantly monitor the

phase changes in applications, and predict the optimal core allocation once a phase change is detected. OptiCore employs two techniques in REEact to detect phase changes. Which of these two techniques is used, depends on the thread libraries used by the multi-threaded applications.

For applications which use OpenMP, the start of a new phase is usually equivalent to the start of new parallel region (usually a new data-parallel loop) [21]. Because a parallel region typically begins with a call to particular function which partitions and schedules a task to concurrent threads, we can detect the start of this new parallel region, or the new phase, by intercepting the invocation of this function. For example, in the GNU OpenMP implementation (GOMP), this function is "GOMP_parallel_start" [146]. OptiCore uses REEact's function-interception service to detect the invocation of "GOMP_parallel_start". It is worth nothing that an OpenMP application may repeatedly executes its parallel regions. Because a parallel region always runs the same code, its memory behavior and optimal core allocation are consist across executions. Consequently, there is no need to re-profile a parallel region after its first execution. OptiCore internally maintains records of the optimal core allocations of executed parallel regions. If a parallel region has been executed before, OptiCore directly executes it with its recorded optimal core allocation without re-profiling it. In this way, OptiCore reduces the overhead associated with online profiling.

For applications which use POSIX Threads (Pthreads), there is a no clear indication of a new phase like OpenMP [65]. For these applications, OptiCore periodically checks their memory behavior by querying the hardware Performance Monitoring Units (PMUs) readings with REEact. If the difference of two consecutive readings is larger than a pre-defined threshold, then a new phase is deemed to have started. On the AMD platform with Opteron 6174 processors, we use the PMU, $SYSTEM\_READ\_RESPONSES$, to monitor the changes in memory behavior. On the Intel platform with X7550 processors, we use the PMU of $OFFCORE\_RESPONSE\_0$. Note that there are more sophisticated phase change detection techniques for Pthreads applications [129, 133]. However, we observed that simple periodical checks were sufficient for the optimal core allocation prediction of our benchmarks.

## 8.3 Experimental Evaluation

This section presents the experimental evaluation of the OptiCore run-time system. In this chapter, we focus on evaluating the performance of OptiCore.

### 8.3.1 Experiment Setup

We evaluated OptiCore on our Intel and AMD NUMA platforms. A detailed description of the Intel and AMD platforms can be found in Section 5.5.1. We conducted experiments with benchmarks from PARSEC and NPB benchmark suites [15, 75]. Two PARSEC benchmarks, *raytrace* and *x264*, experienced segmentation faults when executed with more than 64 threads, so they were excluded from our experiments. PARSEC's native input sets were used for PARSEC benchmarks, except for *swaptions*. For *swaptions*, we used a input set that was twice the size of its native input set so that the input size is large enough to support 256 threads. NPB's *D* input sets were used for NPB benchmarks. These input sets are large enough to fully utilize the large number of cores and threads used in our experiments. NPB benchmark *dc* was excluded from our experiments because it does not have *D* input set. The information of compiler and compilation flags can be found in Section 5.5.1.

### 8.3.2 Evaluation Goals and Evaluation Metric

We run each benchmark with three configurations on both platforms.

1. In the first configuration, which is called the "baseline", each benchmark is executed using all cores with one thread per core under the management of the default PThreads and GOMP libraries. This configuration gives the performance that users can get using current thread libraries and the common use-all-cores allocations.

2. In the second configuration, which is called the "dist-sync", each benchmark is executed using its optimal core allocation with one thread per core and distributed synchronization. This second configuration can be viewed as a second baseline, because it gives the best performance that users can get without any run-time management from OptiCore, including run-time profiling, online optimal core allocation prediction and dynamic core reallocation. Note that, for

multi-phased benchmarks, where each phase has different optimal core allocations, the optimal core allocation of the longest phase is used. The optimal core allocation for each benchmark can be found in Section 5.5.

3. In the third configuration, or the OptiCore configuration, each benchmark is executed with OptiCore using 256 threads. We chose 256 threads, because they provide less-than-5%-overhead guarantee on both NUMA platforms (based on our analysis in Section 6.6). This third configuration gives the actual performance that users can expect by using OptiCore.

Comparing the performance of the three configurations answers the following two questions:

1. What is the overall performance benefit of OptiCore over the current way of executing multi-threaded applications, which uses use-all-cores allocation and default thread libraries? We answer this question by comparing the first and third configurations.

2. What is the overhead of OptiCore run-time system? We answer this question by comparing the performance of the second and third configurations.

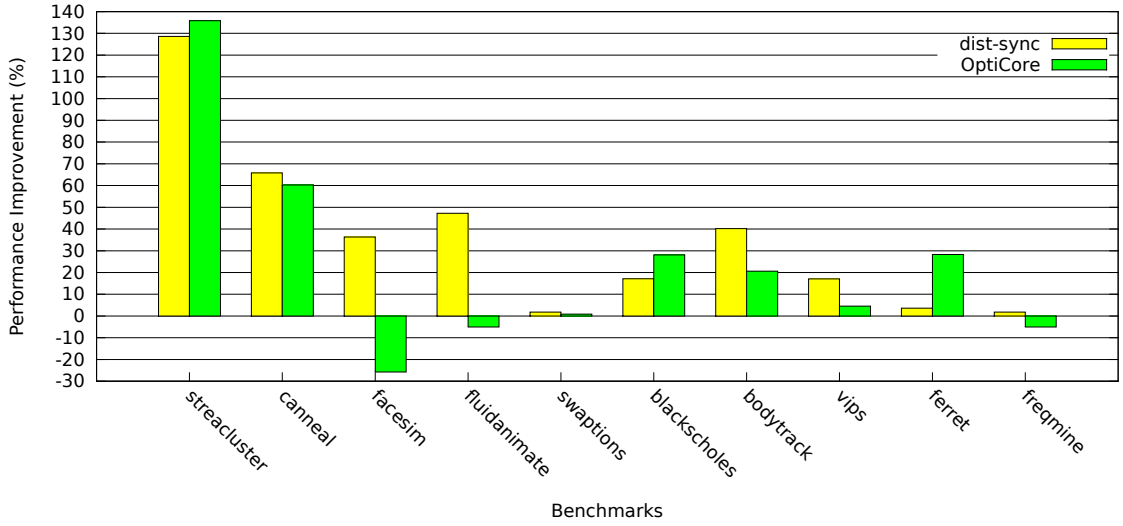
We run each benchmark with each configuration for five trials. We then computed the average execution time for that benchmark under that configuration. We report the performance improvement of one configuration over another. More specifically, for any two configurations, $conf1$ and $conf2$, the performance improvement of $conf2$ over $conf1$ is defined as,

$$Perf\_Improv = \frac{Time_{conf1} - Time_{conf2}}{Time_{conf2}} \times 100\%. \tag{8.1}$$

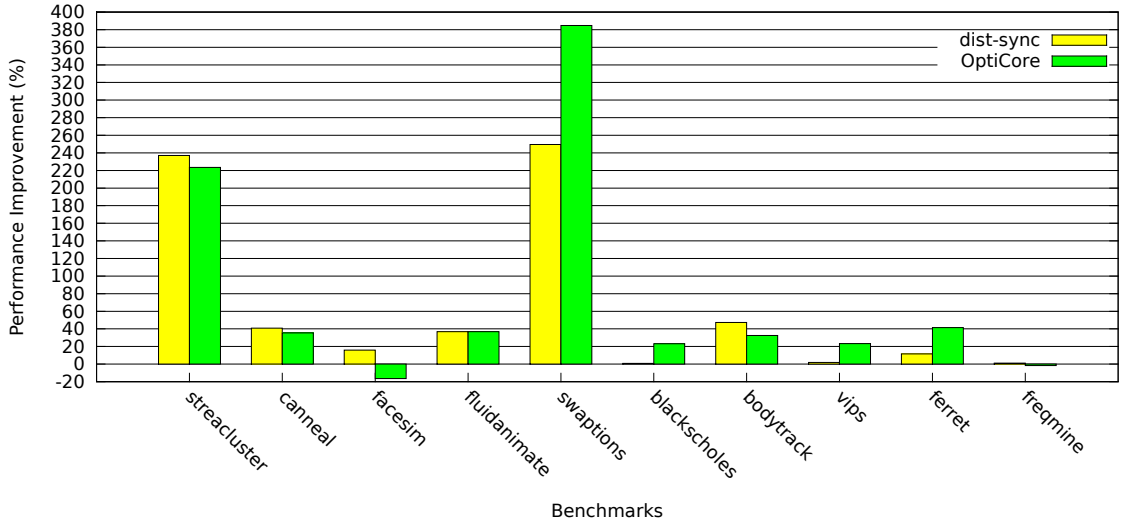
Positive values of $Perf\_Improv$ indicate performance improvement, and negative values indicate slowdown (or overhead).

### 8.3.3 The Performance Benefit of OptiCore

Figure 8.3 and Figure 8.4 gives the performance improvement of the "dist-sync" and OptiCore configurations over the "baseline" configuration. As both figures show, OptiCore performs significantly better than the "baseline" configuration, which uses all-cores and default thread libraries. The highest performance improvement is 384.7%,

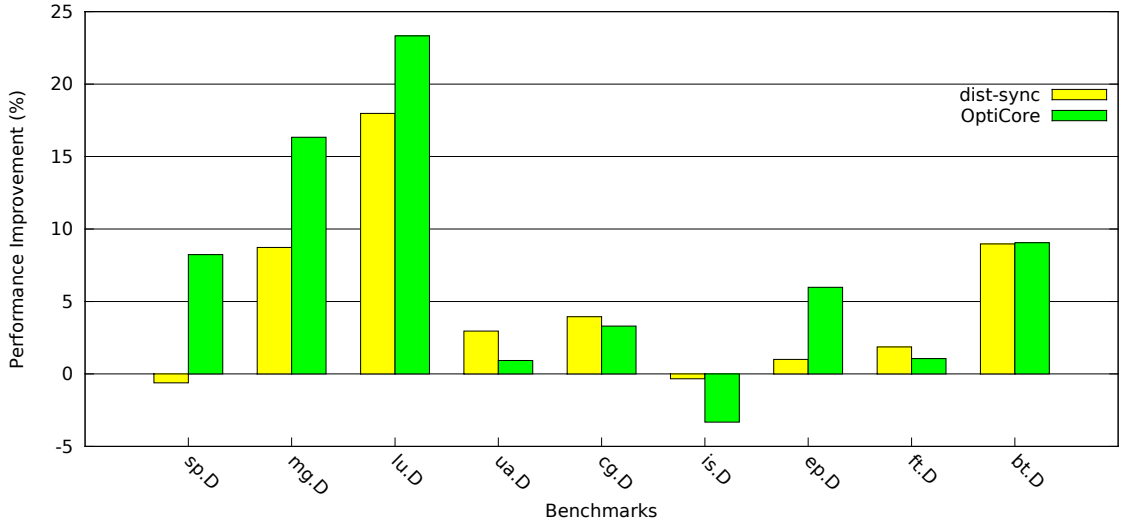



(a) Results on Intel Platform

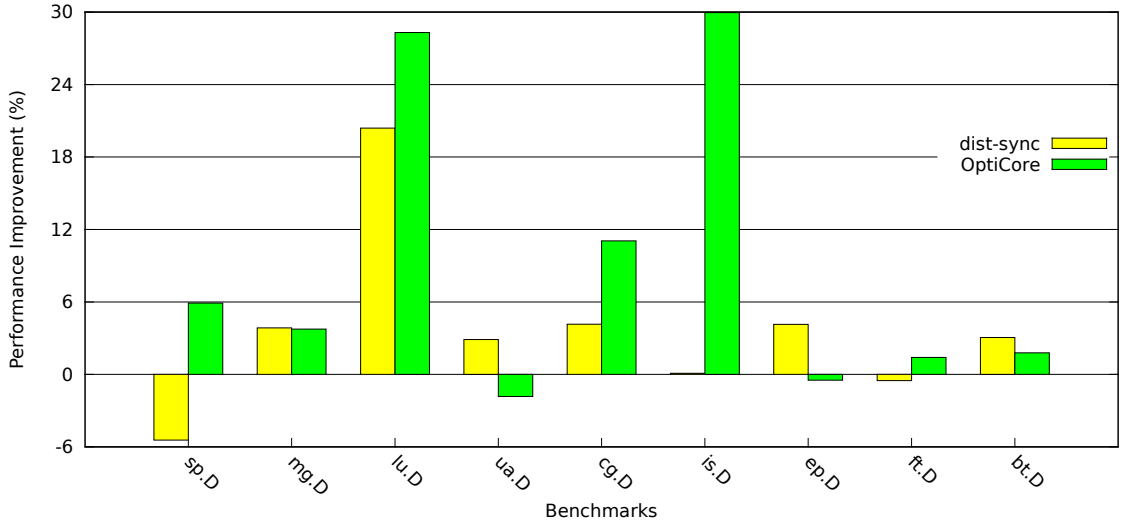



(b) Results on AMD Platform

Figure 8.3: Performance improvement of the "dist-sync" and OptiCore configurations over the "baseline" configuration for PARSEC benchmarks on Intel and AMD NUMA platforms. Positive values indicate speedup while negative values indicate slowdown.




(a) Results on Intel Platform




(b) Results on AMD Platform

Figure 8.4: Performance improvement of the "dist-sync" and OptiCore configurations over the "baseline" configuration for NPB benchmarks on Intel and AMD NUMA platforms. Positive values indicate speedup while negative values indicate slowdown.

which was achieved with benchmark *swaptions* on the AMD platform. The average performance improvement of OptiCore was 51.3% for the PARSEC benchmarks, and 8.0% for the NPB benchmarks. The overall performance improvement of OptiCore was 34.6% on average. This performance benefit of OptiCore is the result of the combined performance improvements from better core allocations, faster (distributed) synchronizations, lower L1 data cache misses, higher processor utilization and better load balancing. The exact improvement from optimal core allocation can be found in Section 5.5. The exact improvement from distributed synchronizations, lower L1 data cache misses, higher processor utilization and better load balancing can be found in Section 6.8.

It is worth noting that, for many benchmarks, the performance improvement is achieved with fewer cores allocated. On the Intel platform, *facesim* has smallest optimal core allocation of all benchmarks, which is 8,8,0,0, or 50% of a total of 32 cores. On the AMD platform, *streamcluster* has the smallest optimal core allocation, which is 6,0,0,0,0,0,0,0, or only 12.5% of a total of 48 cores. On average, OptiCore only allocates 88.7% of all cores, while achieving an average performance improvement of 34.6%.

For *fluidanimate*, *freqmine*, *is.D* and *ep.D*, their performance under OptiCore was slower than the "baseline" configuration, because of the run-time overhead from the profiling and optimal core allocation predictions. However, the slowdowns of these benchmarks are very small, and are always less than 5%. *Facesim* experienced higher than 5% slowdown, because of the negative impact from current Linux scheduler, as discussed in Section 6.8. We plan to address this scheduler in the future.

### 8.3.4 The Run-time Overhead of OptiCore

To determine the run-time overhead of OptiCore, we compare the performance of the configurations of "dist-sync" and OptiCore in Figure 8.3 and Figure 8.4. Both configurations execute their benchmarks with optimal core allocations. However, unlike the static "dist-sync" configuration, OptiCore includes run-time profiling, online optimal core allocation prediction and dynamic core reallocation. As a result, by comparing the performance of the configurations of "dist-sync" and OptiCore, we can determine the overhead of these run-time operations.

As Figure 8.3 and Figure 8.4 show, the maximum slowdown of OptiCore is usually less than 5%. Only four benchmarks, *fluidanimate*, *bodytrack*, *vips* and *facesim* have

higher than 5% overhead, because current Linux scheduler cannot properly schedule large numbers of threads, as discussed in Section 6.8.

Besides slowdown, the run-time operations from OptiCore can also improve performance over the static configuration of "dist-sync". Ten benchmarks benefits from executing large numbers of threads. Because the performance benefit of large numbers of threads has been discussed in Section 6.8, we do not elaborate on it. In addition to the performance improvement of executing large numbers of threads, OptiCore can also dynamically adjust its optimal core allocations to meet the needs of different phases, which further improves the performance of OptiCore over the static execution configuration of "dist-sync". This performance improvement can be observed in the cases of *sp.D* when it executed on the Intel and AMD platforms, as well as the case of *mg.D* on the Intel platform. The performance improves for these three cases were 8.2%, 10.72% and 6.5%, respectively.

Overall, OptiCore performs similarly to the static configuration of "dist-sync". The average overhead of OptiCore is only 1.85% for all PARSEC and NPB benchmarks. If the negative impact from Linux scheduler is excluded from the results, OptiCore is actually 4.78% faster than the static execution configuration. These results suggest that not only does a run-time core allocation manager, such as OptiCore, has little overhead, it may potentially be more beneficial than any static execution scheme because its performance benefits.

## 8.4 Summary

This chapter presents the OptiCore run-time system. OptiCore combines DraMon, NuCore, FlexThread and the REEact framework to automatically execute multi-threaded applications with their optimal core allocations. OptiCore is carefully designed so that no user-involvement or source code is required. We evaluated the performance of OptiCore on two large-scale NUMA platforms using PARSEC and NPB benchmarks. The experimental results show that, by executing with optimal core allocations, the applications managed with OptiCore perform 34.6% faster than the use-all-cores allocation on average. More importantly, the speedup from OptiCore is achieved with fewer cores allocated. The minimal optimal core allocation uses only 12.5% of all cores and performs 223.4% better than using all cores. On average, OptiCore allocates only 88.7% of all cores and achieves 34.6% performance improvement. The experimental results also show that OptiCore has only 1.85% overhead compared

to manually executing each benchmark with its optimal core allocation. These results suggesting that OptiCore offers a complete, low-overhead, user-involvement-free and application-source-code-independent solution for the processor over-provisioning problem on large-scale NUMA platforms.

# Chapter 9

# Summary and Future Work

This chapter summarizes this dissertation and presents directions for future exploration.

## 9.1 Summary of Dissertation

Because of the difficulty of increasing single core performance without significantly increasing processor power consumption, modern micro-architectures have switched to multi-core processors and thread-level parallelism for performance growth. To meet the ever growing need for speed, future large-scale computing platforms will be equipped with dozens, hundreds or even thousands of cores. The applications executing on these platforms usually have good parallelism. They can create large numbers of threads to simultaneously execute on the massive numbers of cores.

When executing these multi-threaded applications on these large-scale NUMA platforms, users typically allocate all cores to their applications, assuming more cores translate to better performance. However, because of limited memory bandwidth, the performance of many multi-threaded applications benefits little from the massive numbers of cores on these large scale systems. In fact, for many multi-threaded applications, allocating all cores, or over-provisioning processors, can only degrade performance, reduce energy efficiency and system throughput. Therefore, it is desirable to execute multi-threaded applications with the minimum core allocation that achieves best performance. We call this allocation, the optimal core allocation. However, because the optimal core allocation varies with application, input set and hardware configuration, determining the optimal core allocations for various multi-threaded applications, and executing these applications with their optimal core allocations, are

very challenging.

### 9.1.1 The Prediction of Optimal Core Allocation

Given that optimal core allocation is primarily determined by limited memory bandwidth, predicting optimal core allocation requires accurate modeling of the memory system of large-scale NUMA machines. To understand how memory bandwidth limitation impacts the optimal core allocations on large-scale NUMA machines, we conducted a series of experiments using PARSEC, NPB and BLAS benchmarks on two large-scale NUMA machines [7, 15, 75]. The experimental results are reported in Chapter 3. These results indicate that there are four major factors that impacts the optimal core allocation. These factors are,

1. the memory bandwidth usage of local DRAM modules;
2. the memory bandwidth usage of inter-node connections;
3. the interference among local and inter-node memory accesses;
4. the heterogeneity within large-scale NUMA machines and multi-threaded applications, including the heterogeneous inter-node connections, the non-uniform inter-node connection topologies, and the varying application memory behaviors and communication patterns.

In short, to accurately model the memory system of a NUMA machine, these factors must be carefully considered. In the first half of this research, we investigated how to accurately model the memory system and predict the optimal core allocations.

#### Modeling the Local DRAM Factor

For the factor of local DRAM bandwidth usage, both our experiments and previous research results indicate that the local DRAM bandwidth is determined by DRAM contention and DRAM request overlapping [72]. However, modeling DRAM contention and request overlapping is very challenging, because they appear to be completely random, and there is no existing statistical distribution can accurately express them.

Unlike previous research which treats DRAM as a black box, we inspected the DRAM internal operations using PMUs [30, 60, 81, 86, 144, 164]. We then implemented simulators to reproduce the DRAM contention and request overlapping that

we observed in real DRAMs. Thanks to this analysis, we discovered that, because of the memory inter-leaving schemes and the large size of DRAM banks, DRAM contention and request overlapping have a stable statistical distribution which can be described with a few thousands of probability equations that loosely resemble binomial distributions. This discovery allows us to design and implement a DRAM model called DraMon to predict the DRAM contention, DRAM request overlapping, and the resulting local memory bandwidth usage. Experimental results using PARSEC and NPB benchmarks on two large-scale NUMA platforms show that DraMon model has 93.4% accuracy on average (Chapter 4).

**Modeling Other Factors and Predicting the Optimal**

Because of inter-node connections are package-based networks, the factors of inter-node connections and local/inter-node interference can be described using various constraint functions (Chapter 5). However, the factor of heterogeneity dramatically increases the complexity of modeling the memory system and predicting the optimal core allocation, as the heterogeneity requires us to consider each node and each memory connection differently. Consequently, because of the heterogeneity, we have to determine the optimal core allocation from a very large solution space (millions of possible core allocations) with thousands of linear and non-linear constraints. The large solution space and large numbers of constraints make it very challenging to determine the optimal solution in a short amount of time.

However, we observe that both the optimal core allocation prediction and memory system modeling are integer problems. Leveraging this integer property, we designed a novel technique that could express non-linear memory system constraints with linear functions. Additionally, we can also express the prediction of the optimal core allocation as a linear objective function which maximizes total memory bandwidth while minimizing the size of the core allocation. With these linear objective function and constraints, we then employ Mix Integer Programming (MIP) to find the optimal solution (optimal core allocation) or predict the memory bandwidth usage with short amount of time.

We summarized these findings in a model called NuCore, and implemented it (Chapter 5). We evaluated the accuracy of NuCore on two large-scale NUMA platforms using PARSEC and NPB benchmarks. The experimental results show that NuCore is highly accurate. More specifically, the optimal core allocations predicted by NuCore differ at most one core per node with real optimal core allocations. Ad-

ditionally, NuCore can predict the total memory bandwidth usage for multi-thread applications with 90% accuracy on average.

### 9.1.2 Automatic Execution using Optimal Core Allocation

The DraMon model and NuCore model enable the prediction of optimal core allocation with high accuracy for multi-threaded applications on large-scale NUMA platforms. However, because both models require information of an application's memory behavior and hardware configuration at run-time, these models cannot directly benefit ordinary users for two reasons. First, the requirement of the information on run-time memory behavior implies that the optimal core allocation is unknown at the beginning of application execution. Therefore, an efficient technique is necessary to dynamically adapt an application to its optimal core allocation during execution. Second, this run-time information can be difficult for non-expert users to acquire. In the second half of our research we focused on supporting dynamic adaption to optimal core allocations, as well as run-time memory behavior profiling and hardware configuration detection. The final goal was to design a run-time system that can automatically execute multi-threaded applications with their optimal core allocations on large-scale NUMA platforms.

#### Efficient Dynamic Adaptation

The major difficulty of dynamic adaptation to optimal core allocations is the load-balancing of any core allocations. Traditionally, users balance the loads of cores by controlling the number of threads. Typically, users create one thread per core to ensure each core has the same load – one thread. Additionally, it is either impossible or heavy-handed to change thread count during execution, i.e., the thread count cannot be changed during execution. However, because the optimal core allocation is determined during execution, it is very difficult to determine the proper number of threads to create before execution. Because of this difficulty, previous research is either limited to data parallel loops (which can be easily repartitioned), or requires source code modification [86, 104, 138].

However, we observe that if an application creates a large number of threads, it is possible to remap these threads to any core allocation to achieve near-ideal load-balancing. That is, creating large numbers of threads allows near-ideal load-balancing for any optimal core allocation without the need of modifying application's

source code. The downside of creating large numbers of threads is that it increases the number of synchronization operations, which in-turn introduces significant overhead. To reduce this overhead, we employ distributed synchronization primitives.

Combining the large numbers of threads and distributed synchronization, we implemented a run-time system, FlexThread, to support efficient dynamic adaptation to any optimal core allocations without application source code modification (Chapter 6). Experimental results with PARSEC and NPB benchmarks on our two large-scale NUMA platforms show that FlexThread has less than 5% overhead, compared to manual execution with one thread per core for different optimal core allocations. This low overhead suggesting that FlexThread can efficiently support dynamic adaption to optimal core allocations. The experimental results also show that, for many benchmarks, FlexThread performs better than using one-thread-per-core due to better CPU and private cache utilization. The flexibility of supporting varying core allocations and the performance benefit suggest that users should always create large numbers of threads for their applications.

**Supporting Run-time Profiling and HW/SW Management**

To support run-time memory behavior profiling and hardware configuration detection, we designed a run-time system framework called REEact (Chapter 7). REEact supports monitoring application memory behavior through the reading of hardware Performance Monitoring Units (PMU). REEact also supports the detection of hardware configuration from the Operating System and hardware registers. Additionally, because FlexThread requires dynamic management of thread execution, REEact also provides means to hook into executing applications to take over the control of its threads. Moreover, REEact provides a wide range of services to support run-time profiling, application management and hardware resource configuration for run-time managements problems other than the optimal core allocation problem. The services provided by REEact require no user-involvement during execution or any knowledge of application source-code. Additionally, REEact has limited overhead. Experimental results on two large-scale NUMA machines with PARSEC benchmarks show that REEact has a maximum overhead of only 3%.

#### Automatic Execution using Optimal Core Allocation

Combing DraMon, NuCore, FlexThread and REEact, we implemented the OptiCore run-time system, which can automatically execute multi-threaded applications with their optimal core allocations (Chapter 8). We evaluated OptiCore using PARSEC and NPB benchmarks on our two large-scale NUMA platforms. The experimental results show that OptiCore can efficiently execute multi-threaded applications with their optimal core allocations. By executing with optimal core allocations, applications managed by OptiCore perform 34.6% faster than the use-all-cores allocation on average. More importantly, the speedup from OptiCore is achieved with fewer cores allocated. The minimal optimal core allocation uses only 12.5% of all cores and performs 223.4% better than using all cores. On average, OptiCore allocates only 88.7% of all cores and achieves 34.6% performance improvement. The experimental results also show that OptiCore has only 1.25% overhead compared to manually executing each benchmark with its optimal core allocation. These results suggesting that OptiCore offers a complete, low-overhead, user-involvement-free and application-source-code-independent solution for the processor over-provisioning problem on large-scale NUMA platforms.

## 9.2 Future Directions

Although this research provides a complete solution to the processor over-provisioning problem, there are still many opportunities to improve the design of OptiCore, or to apply our insights and models to further improve the execution of multi-threaded applications on large-scale NUMA platforms. This section briefly discusses these opportunities.

### 9.2.1 Improving OptiCore Design

#### Improving Linux Thread Management

As discussed in Chapter 6, current Linux scheduling policy does not support large numbers of threads with multiple mutexes and frequent mutex locking attempts. The current Linux scheduler tends to schedule lock-waiting threads to preempt lock-holding threads. This preemption usually ends with the lock-waiting threads quick relinquish the processor, because they cannot acquire the locks. A more efficient way

of scheduling should not allow lock-holding threads to be preempted. We plan to add a new scheduling policy to Linux kernel to improve its support for executing large number of threads.

Additionally, we currently uses FUTEX system calls in Linux to suspend threads for synchronization [53]. However, the FUTEX system call is quite heavy in that is requires locking a global table to add the information of the suspending threads. We plan to improve this suspension mechanism either by breaking the global table into local (per-core) tables, or simply by letting suspending threads yield their processors.

### 9.2.2 Considering Caches, Parallelism and Prefetchers

Although for large-scale applications with large input sets, memory bandwidth is the primary limitation of scalability, the scalability of smaller applications with smaller inputs may be mainly limited by cache size, cache contentions or parallelism. To handle these applications with OptiCore, we plan to incorporate the models and techniques from previous work into OptiCore in the future [25, 54, 56, 60, 79, 86, 99, 123, 132, 144, 159].

Furthermore, our models currently do not consider the aggressive prefetchers on Intel processors. We will investigate the behavior of these prefetchers and extend our models to predict the scalability impact of these prefetchers.

### 9.2.3 Beyond Optimal Core Allocation for Better Performance

Through the investigation of this research, we have gathered valuable insights about the large-scale NUMA platforms and multi-threaded applications. We have also designed highly accurate models for memory systems. These insights and models behoove us to apply them to problems beyond the optimal core allocation and performance.

#### Energy Efficiency and System Throughput

In addition to the negative performance impact, processor over-provisioning can also reduce energy efficiency and system throughput, because the extra cores allocated can be turned off to save energy or be used to execute other applications to improve overall system throughput. However, both energy efficiency and system throughput

are more complex optimization goals than execution time. Allocating an extra core may slightly reduce execution time. However, this extra core may greatly increase power consumption which makes the whole execution less energy efficient. Similarly, to improve system throughput, it is important that we balance the core allocations for multiple applications so that the overall throughput is maximized. For optimal energy efficiency and system throughput, new models that considers more hardware and software properties must be invented.

### Scalable Memory Access

This project convinces us memory bandwidth will remain the primary scalability limitation for the foreseeable future. The problem is not simply that hardware does not provide enough memory bandwidth. The analysis in Chapter 4 has revealed that there is a huge inefficiency in the way that software utilizes memory resources: current programs can only utilize less than 60% of available bandwidth at best. Although future DRAM modules will provide higher bandwidth, there is a good chance that most would be is wasted. Because this inefficiency is the result of both software memory access pattern and memory allocation scheme, we believe that addressing this problem requires rethinking the way memory is accessed and designing more flexible memory allocation schemes that suit the varying needs of real applications. More specifically, our goal is to investigate how to use DRAM information to rearrange memory accesses in application algorithms and improve memory allocation to reduce bank-level contention and improve scalability.

### Configurable Hardware with Program Hint

In this research, we constantly observe that the efficiency of different hardware component varies considerably with application. The problem is that the configuration or algorithm used by a hardware component is generic which does not always meet the need of every application. As the architecture going into the Dark Silicon era, a great portion of a processor can be devoted to configurable hardware components that support different algorithms and designs. We believe compile-time and run-time information from applications should be used to guide the dynamic configuration of the underlying hardware.

# Bibliography


[1] "GNU GOMP libgomp Documentation", 2014.

[2] T. Achterberg. SCIP: solving sonstraint integer programs. *Mathematical Programming Computation*, 1(1), 2009.

[3] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative Task Management Without Manual Stack Management. In *USENIX Annual Technical Conference*, 2002.

[4] J. H. Ahn, M. Erez, and W. J. Dally. The Design Space of Data-Parallel Memory Systems. In *Int'l Conf. on Supercomputing*, 2006.

[5] W. Y. Alkowaileet. *NUMA-aware multicore Matrix Multiplication.* Master's thesis, University of California, Irvine, 2013.

[6] AMD. BIOS and Kernel Developer's Guide (BKDG) For AMD Family 10h Processors, 2013.

[7] AMD. AMD Core Math Library (ACML), 2013.

[8] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. In *Proc. of the Thirteenth ACM Symposium on Operating Systems Principles*, 1991.

[9] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism. *ACM Transactions on Computer Systems (TOCS)*, 1992.

[10] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, et al. The Landscape

of Parallel Computing Research: A View from Berkeley. Technical report, University of California, Berkeley, 2006.

[11] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian, and A. Davis. Handling the Problems and Opportunities Posed by Multiple On-chip Memory Controllers. In *Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2010.

[12] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. of the Nineteenth ACM Symposium on Operating Systems Principles*, 2003.

[13] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proc. of the 22nd ACM Symposium on Operating Systems Principles*, 2009.

[14] J. Beirlant, Y. Goegebeur, J. Segers, and J. Teugels. *Statistics of Extremes: Theory and Applications*. John Wiley & Sons, 2006.

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

[16] C. Bienia and K. Li. Fidelity and Scaling of the PARSEC Benchmark Inputs. In *Int'l Symp. on Workload Characterization*, 2010.

[17] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In *Int'l. Conf. on Parallel Architectures and Compilation Techniques*, 2008.

[18] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28, 2002.

[19] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali. A case for numa-aware contention management on multicore systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, 2010.

[20] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova. A Case for NUMA-aware Contention Management on Multicore Systems. In *Proceedings of USENIX Annual Technical Conference*, 2011.

[21] O. A. R. Board. OpenMP Application Program Interface, 2013.

[22] M. Bohr. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1), 2007.

[23] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[24] E. Castillo. *Extreme Value Theory in Engineering.* Elsevier, 1988.

[25] G. Chadha, S. Mahlke, and S. Narayanasamy. When Less Is MOre (LIMO):Controlled Parallelism for Improved Efficiency. In *Proceedings of the Int'l Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, 2012.

[26] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *Int'l Symp. on High-Performance Computer Architecture*, 2005.

[27] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. P. Jouppi. Staged Reads: Mitigating the Impact of DRAM Writes on DRAM Reads. In *Int'l Symp. on High Performance Computer Architecture*, 2012.

[28] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *1st Workshop on Architectural and System Support for Improving Software Dependability*, 2006.

[29] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson. Scheduling threads for constructive cache sharing on CMPs. In *ACM Symp. on Parallel Algorithms and Architectures*, 2007.

[30] H. Choi, J. Lee, and W. Sung. Memory Access Pattern-Aware DRAM Performance Model for Multi-core Systems. In *Int'l Symp. on Performance Analysis of Systems and Software*, 2011.

[31] J. Corbalán, X. Martorell, and J. Labarta. Performance-Driven Processor Allocation. In *Int'l Conference on Symposium on Operating System Design & Implementation*, 2000.

[32] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online Power-performance Adaptation of Multithreaded Programs Using Hardware Event-based Prediction. In *Proc. of Int'l Conf. on Supercomputing*, 2006.

[33] M. Curtis-Maury, A. Shah, F. Blagojevic, D. S. Nikolopoulos, B. R. de Supinski, and M. Schulz. Prediction Models for Multi-dimensional Power-performance Optimization on Many Cores. In *Proc. of Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2008.

[34] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proc. of the Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2013.

[35] J. del Cuvillo. *Breaking away from the OS Shadow: A Program Execution Model Aware Thread Virtual Machine for Multicore Architectures*. PhD thesis, University of Delaware, Newark, DE, USA, 2008.

[36] J. Demme and S. Sethumadhavan. Rapid Identification of Architectural Bottlenecks via Precise Event Counting. In *Proc. of Int'l Symp. on Computer Architecture*, 2011.

[37] R. Dennard, V. Rideout, E. Bassous, and A. LeBlanc. Design of Ion-implanted MOSFET's with Very Small Physical Dimensions. *IEEE Journal of Solid-State Circuits*, 9(5), Oct 1974.

[38] C. Ding and Y. Zhong. Predicting Whole-program Locality through Reuse Distance Analysis. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2003.

[39] Y. Ding, M. Kandemir, P. Raghavan, and M. J. Irwin. A Helper Thread Based EDP Reduction Scheme for Adapting Application Execution in CMPs. In *Int'l Symp. on Parallel and Distributed Processing*, 2008.

[40] Y. Du, M. Zhou, B. Childers, D. Mosse, and R. Melhem. In *Proc. of Int'l Symp. on High Performance Computer Architecture*, 2015.

[41] K. Du Bois, S. Eyerman, and L. Eeckhout. Per-thread Cycle Accounting in Multicore Processors. *ACM Trans. on Architecture and Code Optimization (TACO)*, 9(4), 2013.

[42] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout. Bottle graphs: Visualizing scalability bottlenecks in multi-threaded applications. In *Proc. of the Int'l Conf. on Object Oriented Programming Systems Languages and Applications*, 2013.

[43] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and Predicting Program Behavior and its Variability. In *Proc. of Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2003.

[44] E. Ebrahimi, O. Mutlu, C. J. Lee, and Y. N. Patt. Coordinated Control of Multiple Prefetchers in Multi-core Systems. In *Proc. of Int'l Symp. on Microarchitecture*, 2009.

[45] D. Eklov, D. Black-Schaffer, and E. Hagersten. Fast Modeling of Shared Caches in Multicore Systems. In *Proc. of Int'l Conf. on High Performance and Embedded Architectures and Compilers*, 2011.

[46] R. S. Engelschall. Portable Multithreadingthe Signal Stack Trick for User-space Thread Creation. In *USENIX Annual Technical Conference*, 2000.

[47] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proc. of the Fifteenth ACM Symposium on Operating Systems Principles*, 1995.

[48] S. Eranian. Perfmon2: A flexible performance monitoring interface for linux. In *Linux Symp.*, 2006.

[49] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark Silicon and the End of Multicore Scaling. In *Proc. of Int'l Symp. on Computer Architecture*, 2011.

[50] H. Esmaeilzadeh, T. Cao, Y. Xi, S. M. Blackburn, and K. S. McKinley. Looking Back on the Language and Hardware Revolutions: Measured power, Performance, and Scaling. In *Proc. of Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[51] Z. Fang, L. Zhang, J. B. Carter, S. A. Mckee, A. Ibrahim, M. A. Parker, and X. Jiang. Active Memory Controller. *The Journal of Supercomputing*, 62(1), 2012.

[52] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proc. of Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2012.

[53] H. Franke, R. Russell, and M. Kirkwood. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of AUUG Conference*, 2002.

[54] S. S. Fu and N.-F. Tzeng. A Circular List Based Mutual Exclusion Scheme for Large Shared-memory Multiprocessors. *IEEE Trans. on Parallel and Distributed Systems*, 8(6), 1997.

[55] S. Ghose, H. Lee, and J. F. Martínez. Improving Memory Scheduling via Processor-side Load Criticality Information. In *Int'l Symp. on Computer Architecture*, 2013.

[56] R. Gupta and C. R. Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. *Int'l Journal of Parallel Programming*, 18 (3), 1989.

[57] J. Haase, A. Hofmann, and K. Waldschmidt. A self distributing virtual machine for adaptive multicore environments. *Int'l Journal of Parallel Programming*, 38 (1), 2010.

[58] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches. In *Proc. of Int'l Symp. on Computer Architecture*, 2009.

[59] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout. Using Cycle Stacks to Understand Scaling Bottlenecks in Multi-threaded Workloads. In *Int'l Symp. Workload Characterization*. IEEE, 2011.

[60] W. Heirman, T. Carlson, K. Van Craeynest, I. Hur, A. Jaleel, and L. Eeckhout. Undersubscribed Threading on Clustered Cache Architectures. In *Proc. of Int'l Symp. on High Performance Computer Architecture*, 2014.

[61] J. L. Hennessy and D. A. Patterson. *Computer Architecture : A Quantitative Approach*. Morgan Kaufmann/Elsevier, 5th edition, 2011. ISBN 012383872X, 9780123838728.

[62] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Elsevier, 2012.

[63] E. Herrero, J. Gonzalez, R. Canal, and D. Tullsen. Thread Row Buffers: Improving Memory Performance Isolation and Throughput in Multiprogrammed Environments. *IEEE Trans. on Computers*, 62(9), 2013.

[64] R. J. Hyndman and A. B. Koehler. Another look at measures of forecast accuracy. *International Journal of Forecasting*, 22(4), 2006.

[65] IEEE. IEEE Std. 1003.1c-1995 thread extensions, 1995.

[66] Intel. An Introduction to the Intel QuickPath Interconnect, 2009.

[67] Intel. Intel 64 and IA-32 architecture software developer's manual, 2009.

[68] Intel. Reference Manual for Intel Math Kernel Library 11.1 Update 3, 2013.

[69] Intel. User and Reference Guide for the Intel C++ Compiler 14.0, 2014.

[70] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *Proc. of the Int'l Symp. on Computer Architecture*, 2008.

[71] M. M. Islam and P. Stenstrom. A Unified Approach to Eliminate Memory Accesses Early. In *Int'l Conf. on Compilers, Architectures and Synthesis for Embedded Systems*, 2011.

[72] B. Jacob, S. Ng, and D. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2010.

[73] W. Jia, K. A. Shaw, and M. Martonosi. MRPB: Memory request prioritization for massively parallel processors. In *Proc. of the Int'l Conf. on High Performance Computer Architecture*, 2014.

[74] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin, and R. Balasubramonian. CHOP: Integrating DRAM Caches for CMP Server Platforms. *IEEE Micro*, 31(1), 2011.

[75] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical report, NASA Ames Research Center, 1999.

[76] L. Jin and S. Cho. SOS: A Software-Oriented Distributed Shared Cache Management Approach for Chip Multiprocessors. In *Int'l. Conf. on Parallel Architectures and Compilation Techniques*, 2009.

[77] C. Jung, D. Lim, J. Lee, and S. Han. Adaptive Execution Techniques for SMT Multiprocessor Architectures. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2005.

[78] M. Kadin, S. Reda, and A. Uht. Central vs. Distributed Dynamic Thermal Management for Multi-core Processors: Which one is Better? In *Proc. of ACM Great Lakes Symp. on VLSI*, 2009.

[79] O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das. Neither More nor Less: Optimizing Thread-level Parallelism for GPGPUs. In *Proc. of Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2013.

[80] D. U. Kim, S. Yoon, and J. W. Lee. An Analytical Model to Predict Performance Impact of DRAM Bank Partitioning. In *Workshop on Memory Systems Performance and Correctness*, 2013.

[81] M. Kim, P. Kumar, H. Kim, and B. Brett. Predicting Potential Speedup of Serial Code via Lightweight Profiling and Emulations with Memory Performance Model. In *Int'l Symp. on Parallel and Distributed Processing Symposium*, 2012.

[82] S. Kim, D. Chandra, and Y. Solihin. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. In *Proc. of Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2004.

[83] T. Koch. *Rapid Mathematical Prototyping*. PhD thesis, Technische Universität Berlin, 2004.

[84] M. Kudlur and S. Mahlke. Orchestrating the Execution of Stream Programs on Multicore Platforms. In *Proc. of Programming Language Design and Implementation*, 2008.

[85] E. Kursun, G. Reinman, S. Sair, A. Shayesteh, and T. Sherwood. Low-Overhead Core Swapping for Thermal Management. In *Workshop on Power-Aware Computer Systems*, 2004.

[86] J. Lee, H. Wu, M. Ravichandran, and N. Clark. Thread Tailor: Dynamically Weaving Threads Together for Efficient, Adaptive Parallel Applications. In *Int'l Symp. on Computer Architecture*, 2010.

[87] H. W. Lenstra Jr. Integer Programming with a Fixed Number of Variables. *Mathematics of Operations Research*, 8(4), 1983.

[88] J. Li and J. F. Martinez. Dynamic Power-performance Adaptation of Parallel Computation on Chip Multiprocessors. In *Int'l Symp. on High-Performance Computer Architecture*, 2006.

[89] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen, and C. Wu. A Software Memory Partition Approach for Eliminating Bank-level Interference in Multicore Systems. In *Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2012.

[90] X. Liu and J. Mellor-Crummey. A Tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures. In *Proc. of the Int'l Symp. on Principles and Practice of Parallel Programming*, 2014.

[91] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2005.

[92] Z. Majo and T. Gross. Memory system performance in a numa multicore multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage (SYSTOR*, 2011.

[93] Z. Majo and T. R. Gross. Memory management in numa multicore systems: Trapped between cache contention and interconnect overhead. In *International Symp. on Memory Management*, 2011.

[94] Z. Majo and T. R. Gross. Memory System Performance in a NUMA Multicore Multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage*, 2011.

[95] Z. Majo and T. R. Gross. (Mis)Understanding the NUMA Memory System Performance of Multithreaded Workloads. In *IEEE Int'l Symp. on Workload Characterization*, 2013.

[96] A. Mandal, M. Y. Lim, A. Porterfield, and R. Fowler. Effects of Multi-core Memory Concurrency Limits on Multi-threaded Applications. Technical report, Renaissance Computing Institute, 2010.

[97] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. First-class User-level Threads. *ACM SIGOPS Operating Systems Review*, 1991.

[98] L. W. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *USENIX Annual Technical Conference*, 1996.

[99] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Trans. on Computer Systems (TOCS)*, 1991.

[100] W. Mi, X. Feng, J. Xue, and Y. Jia. Software-hardware Cooperative DRAM Bank Partitioning for Chip Multiprocessors. In *Int'l Conf. on Network and Parallel Computing*. 2010.

[101] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8), 1965.

[102] R. Moore. *Predicting Application Performance for Chip Multiprocessors*. PhD thesis, University of Pittsburgh, 2014.

[103] R. Moore and B. Childers. Using Utility Prediction Models to Dynamically Choose Program Thread Counts. In *Int'l Symp. on Performance Analysis of Systems and Software*, 2012.

[104] R. W. Moore and B. R. Childers. Inflation and Deflation of Self-adaptive Applications. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2011.

[105] R. W. Moore and B. R. Childers. Automatic Generation of Program Affinity Policies Using Machine Learning. In *Proc. of Int'l Conf. on Compiler Construction*, 2013.

[106] R. W. Moore and B. R. Childers. Program Affinity Performance Models for Performance and Utilization. In *Proc. of Conf. on Design, Automation & Test in Europe*, 2014.

[107] R. W. Moore and B. R. Childers. Building and using application utility models to dynamically choose thread counts. *The Journal of Supercomputing*, 68(3), 2014.

[108] J. Mukundan and J. F. Martinez. MORSE: Multi-objective Reconfigurable Self-optimizing Memory Scheduler. In *Int'l Symp. on High Performance Computer Architecture*, 2012.

[109] J. Mukundan, H. Hunter, K.-h. Kim, J. Stuecheli, and J. F. Martínez. Understanding and Mitigating Refresh Overheads in High-Density DDR4 DRAM Systems. In *Int'l Symp. on Computer Architecture*, 2013.

[110] O. Mutlu and T. Moscibroda. Stall-time Fair Memory Access Scheduling for Chip Multiprocessors. In *Int'l Symp. on Microarchitecture*, 2007.

[111] O. Mutlu and T. Moscibroda. Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems. In *Int'l Symp. on Computer Architecture*, 2008.

[112] P. Nair, C.-C. Chou, and M. K. Qureshi. A Case for Refresh Pausing in DRAM Memory Systems. In *Int'l Symp. on High Performance Computer Architecture*, 2013.

[113] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private machines: A resource abstraction. Technical report, In University of Wisconsin - Madison, ECE TR, 2007.

[114] T. D. Nguyen, R. Vaswani, and J. Zahorjan. Maximizing Speedup through Self-Tuning of Processor Allocation. In *Int'l Conf. on Parallel Processing*, 1996.

[115] D. Nikolopoulos, G. Back, J. Tripathi, and M. Curtis-Maury. VT-ASOS: Holistic system software customization for many cores. In *IEEE Int'l Symposium on Parallel and Distributed Processing*, 2008.

[116] A. Noll, A. Gal, and M. Franz. CellVM: A Homogeneous Virtual Machine Runtime System for a Heterogeneous Single-Chip Multiprocessor. In *Workshop on Cell Systems and Applications*, 2008.

[117] T. Nowatzki, M. Sartin-Tarm, L. De Carli, K. Sankaralingam, C. Estan, and B. Robatmili. A General Constraint-centric Scheduling Framework for Spatial Architectures. In *Proc. of Programming Language Design and Implementation*, 2013.

[118] H. Park, S. Baek, J. Choi, D. Lee, and S. H. Noh. Regularities Considered Harmful: Forcing Randomness to Memory Accesses to Reduce Row Buffer Conflicts for Multi-core, Multi-bank Systems. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems* , 2013.

[119] G. Pataki, M. Tural, and E. B. Wong. Basis Reduction and the Complexity of Branch-and-Bound. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, 2010.

[120] L. Peng, J.-K. Peir, T. K. Prakash, C. Staelin, Y.-K. Chen, and D. Koppelman. Memory Hierarchy Performance Measurement of Commercial Dual-core Desktop Processors. *Journal of Systems Architecture*, 54(8), 2008.

[121] S. H. Pugsley, Z. Chishti, C. Wilkerson, P. fei Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian. Sandbox Prefetching: Safe Runtime Evaluation of Aggressive Prefetchers. In *Proc. of Int'l Symp. on High Performance Computer Architecture*, 2014.

[122] K. Pusukuri, R. Gupta, and L. Bhuyan. No More Backstabbing... A Faithful Scheduling Policy for Multithreaded Programs. In *Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2011.

[123] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Thread Reinforcer: Dynamically Determining Number of Threads via OS Level Monitoring. In *Int'l Symp. on Workload Characterization*, 2011.

[124] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Thread Tranquilizer: Dynamically Reducing Performance Variation. *ACM Trans. on Architecture and Code Optimization (TACO)*, 8(4), 2012.

[125] M. K. Qureshi and G. H. Loh. Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design. In *Int'l Symp. on Microarchitecture*, 2012.

[126] P. Radojković, V. Čakarević, M. Moretó, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero. Optimal Task Assignment in Multithreaded Processors: A Statistical Approach. In *Proc. of Int'l Conf on Architectural Support for Programming Languages and Operating Systems*, 2012.

[127] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the Bandwidth wall: Challenges in and Avenues for CMP Scaling. In *Int'l Symp. on Computer Architecture*, 2009.

[128] Samsung. 240pin Registered DIMM based on 2Gb C-die.

[129] R. Sarikaya, C. Isci, and A. Buyuktosunoglu. Runtime application behavior prediction using a statistical metric model. *IEEE Transactions on Computers*, 62, 2013.

[130] H. Sasaki, T. Tanimoto, K. Inoue, and H. Nakamura. Scalability-Based Manycore Partitioning. In *Proceedings of the Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2012.

[131] W. Scherer III. *Synchronization and Concurrency in User-level Software Systems*. University of Rochester, 2006.

[132] M. L. Scott and J. M. Mellor-Crummey. Fast, Contention-free Combining Tree Barriers for Shared-memory Multiprocessors. *Int'l Journal of Parallel Programming*, 1994.

[133] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 23(6), 2003.

[134] J. Shlens. Notes on Kullback-Leibler Divergence and Likelihood Theory. *Systems Neurobiology Laboratory*, 2007.

[135] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc of Int'l Symp. on Mass Storage Systems and Technologies*, May 2010.

[136] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A Framework for Performance Modeling and Prediction. In *Int'l Conf. on Supercomputing*, 2002.

[137] S. Sridharan, G. Gupta, and G. S. Sohi. Holistic Run-time Parallelism Management for Time and Energy Efficiency. In *Proc. of Int'l Conf. on Supercomputing*, 2013.

[138] S. Sridharan, G. Gupta, and G. S. Sohi. Adaptive, Efficient, Parallel Execution of Parallel Programs. In *Proceedings of Conf. on Programming Language Design and Implementation*, 2014.

[139] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers. In *Proc. of Int'l Symp. on High Performance Computer Architecture*, 2007.

[140] U. Steinberg and B. Kauer. Towards a scalable multiprocessor user-level environment. In *Workshop on Isolation and Integration for Dependable Systems*, 2010.

[141] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John. The Virtual Write Queue: Coordinating DRAM and Last-level Cache Policies. In *Int'l Symp. on Computer Architecture*, 2010.

[142] J. Stuecheli, D. Kaseridis, H. C. Hunter, and L. K. John. Elastic Refresh: Techniques to Mitigate Refresh Penalties in High Density Memory. In *Int'l Symp. on Microarchitecture*. IEEE, 2010.

[143] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems. In *Int'l Symp. on High Performance Computer Architecture*, 2013.

[144] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven Threading: Power-efficient and High-performance Execution of Multi-threaded Workloads on CMPs. In *Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[145] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations. In *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2009.

[146] G. Team. GNU libgomp: GNU Offloading and Multi Processing Runtime Library, 2014.

[147] R. Teodorescu and J. Torrellas. Variation-aware application scheduling and power management for chip multiprocessors. In *Int'l Symp. on Computer Architecture*, 2008.

[148] The HyperTransport Consortium. HyperTransport I/O Technology Overview, 2004.

[149] R. Thomas and K. Yelick. Efficient FFTs on IRAM. In *Workshop on Media Processors and DSPs*, 1999.

[150] J. Tulip, J. Bekkema, and K. Nesbitt. Multi-threaded Game Engine Design. In *Proc. of Australasian Conf. on Interactive Entertainment*, IE '06, 2006.

[151] H. M. Wagner. An integer linear-programming model for machine scheduling. *Naval Research Logistics Quarterly*, 6(2), 1959.

[152] D. T. Wang. *Modern DRAM Memory Systems: Performance Analysis and Scheduling Algorithm.* University of Maryland, 2005.

[153] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. Bigdatabench: A big data benchmark suite from internet services. In *Proc. of Int'l Symp. on High Performance Computer Architecture*, 2014.

[154] W. Wang, T. Dey, J. W. Davidson, and M. L. Soffa. DraMon: Predicting Memory Bandwidth Usage of Multi-threaded Programs with High Accuracy

and Low Overhead. In *Int'l Symp. on High Performance Computer Architecture*, 2014.

[155] V. Weaver, D. Terpstra, and S. Moore. Non-determinism and Overcount on Modern Hardware Performance Counter Implementations. In *Int'l Symp. on Performance Analysis of Systems and Software*, April 2013.

[156] K. B. Wheeler, R. C. Murphy, and D. Thain. Qthreads: An API for Programming with Millions of Lightweight Threads. In *Int'l Symp. Parallel and Distributed Processing*. IEEE, 2008.

[157] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52(4), Apr. 2009.

[158] J. Winter and D. Albonesi. Scheduling Algorithms for Unpredictably Heterogeneous CMP Architectures. In *Proc. of Int'l Conf. on Dependable Systems and Networks With FTCS and DCC*, 2008.

[159] X. Xiang, B. Bao, C. Ding, and Y. Gao. Linear-time Modeling of Program Working Set in Shared Cache. In *Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2011.

[160] X. Xiang, B. Bao, C. Ding, and K. Shen. Cache Conscious Task Regrouping on Multicore Processors. In *Int'l Symp. on Cluster, Cloud and Grid Computing*, 2012.

[161] Y. Xie and G. H. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects*, 2008.

[162] J. Yang, X. Zhou, M. Chrobak, Y. Zhang, and L. Jin. Dynamic Thermal Management through Task Scheduling. In *Proc. of Int'l Symp. on Performance Analysis of Systems and Software*, 2008.

[163] I. Yeo, C. C. Liu, and E. J. Kim. Predictive Dynamic Thermal Management for Multicore Systems. In *Proc. of the Annual Design Automation Conference*, 2008.

[164] G. L. Yuan and T. M. Aamodt. A Hybrid Analytical DRAM Performance Model. In *Workshop on Modeling, Benchmarking and Simulation*, 2009.

[165] E. Z. Zhang, Y. Jiang, and X. Shen. Does Cache Sharing on Modern CMP Matter to the Performance of Contemporary Multithreaded Programs? In *Proc. of the ACM Symposium on Principles and Practice of Parallel Programming*, 2010.

[166] X. Zhang, S. Dwarkadas, and K. Shen. Hardware Exxecution Throttling for Multi-core Resource Management. In *Proc. of USENIX Annual Technical Conference*, 2009.

[167] X. Zhang, Y. Zhang, B. R. Childers, and J. Yang. Exploiting dram restore time variations in deep sub-micron scaling. In *Proc. of Conf. on Design, Automation & Test in Europe*, 2015.

[168] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2010.

[169] S. Zhuravlev, S. Blagodurov, and A. Fedorova. AKULA: a toolset for experimenting and developing thread placement algorithms on multicore systems. In *Proc. of the 19th Int'l Conf. on Parallel Architectures and Compilation Techniques*, 2010.