Energy-aware Thread and Data Management in Heterogeneous Multi-Core, Multi-Memory Systems

Chun-Yi Su

Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science & Application

December 16, 2014
Blacksburg, Virginia

Keywords: Thread Management, Multi-core Processors, Performance Modeling and Analysis, Power-Aware Computing, Heterogeneous Memory, Data Management

Energy-aware Thread and Data Management in Heterogeneous Multi-Core, Multi-Memory Systems

Chun-Yi Su

ABSTRACT

By 2004, microprocessor design focused on multicore scaling—increasing the number of cores per die in each generation—as the primary strategy for improving performance. These multicore processors typically equip multiple memory subsystems to improve data throughput. In addition, these systems employ heterogeneous processors such as GPUs and heterogeneous memories like non-volatile memory to improve performance, capacity, and energy efficiency.

With the increasing volume of hardware resources and system complexity caused by heterogeneity, future systems will require intelligent ways to manage hardware resources. Early research to improve performance and energy efficiency on heterogeneous, multi-core, multi-memory systems focused on tuning a single primitive or at best a few primitives in the systems. The key limitation of past efforts is their lack of a holistic approach to resource management that balances the tradeoff between performance and energy consumption. In addition, the shift from simple, homogeneous systems to these heterogeneous, multicore, multi-memory systems requires in-depth understanding of efficient resource management for scalable execution, including new models that capture the interchange between performance and energy, smarter resource management strategies, and novel low-level performance/energy tuning primitives and runtime systems. Tuning an application to control available resources efficiently has become a daunting challenge; managing resources in automation is still a dark art since the tradeoffs among programming, energy, and performance remain insufficiently understood.

In this dissertation, I have developed theories, models, and resource management techniques to enable energy-efficient execution of parallel applications through thread and data management in these heterogeneous multi-core, multi-memory systems. I study the effect of dynamic concurrent throttling on the performance and energy of multi-core, non-uniform memory access (NUMA) systems. I use critical path analysis to quantify memory contention in the NUMA memory system and determine thread mappings. In addition, I implement a runtime system that combines concurrent throttling and a novel thread mapping algorithm to manage thread resources and improve energy efficient execution in multi-core, NUMA systems.

In addition, I propose an analytical model based on the queuing method that captures important factors in multi-core, multi-memory systems to quantify the tradeoff between performance and energy. The model considers the effect of these factors in a holistic fashion that provides a general view of performance and energy consumption in contemporary systems.

Finally, I focus on resource management of future heterogeneous memory systems, which may combine two heterogeneous memories to scale out memory capacity while maintaining reasonable power use. I present a new memory controller design that combines the best aspects of two baseline heterogeneous page management policies to migrate data between two heterogeneous memories so as to optimize performance and energy.

DEDICATION

This dissertation is dedicated to my wife: ChiaFang and my daughter: Theresa. My wife’s love, patience, support and understanding have lightened up my spirit to finish this dissertation.

ACKNOWLEDGEMENTS

This dissertation was completed with a lot of support of my family, academic advisors and fellows. Without them, I could not finish the dissertation.

First and foremost, I would like to thank my wonderful wife, ChiaFang. She took care of most of our family chores so that I could focus on my work during the years. Her support, encouragement, quiet patience and unwavering love were undeniably the bedrock upon which the past ten years of my life have been built. Her tolerance of my occasional vulgar moods is a testament in itself of her unyielding devotion and love. I am happy that our lives will move on to the next chapter soon after the end of this work. I thank my parents, HsinYi and YanShu, for their faith in me and allowing me to study aboard for a long time.

I would also like to thank Dr. Kirk W. Cameron for his assistance and guidance in getting my graduate career started on the right foot. I thank him for his wisdom and his understanding about my difficulties during my graduate time. He taught me how to survive an intensive Ph.D. program and guided me to be on the right path of being a researcher. I am also in appreciation of Dr. Dimitrios S. Nikolopoulos. I learned a lot from him about the attitude of doing a good research. I thank him for many insightful conversations as well as for the many helpful comments carried on long-distance via e-mail during the development of my several papers. I also owe a lot of thanks to Dr. Edgar A. León during the time I was at LLNL. He supervised me to finish the third part of the dissertation and looked for much software/hardware resource for me to finish the research. Without him, I could not have made progress as fast as I did. I would also like to thank my other Ph.D. committee members, Dr. Eli Tilevich and Dr. Ali Butt for their discussions and suggestions of this dissertation. I specially thank my committee member Dr. Dong Li, as well as a great friend for his patience and insightful advice while at Virginia Tech. I also thank Dr. Bronis R de Supinski, who gave me support and help to revise my proposals and papers during the time of LLNL. I thank Dr. Gabriel Loh and Dr. David Roberts from AMD Research. I learned decent knowledge of memory simulation from them via our bi-weekly meeting. I cannot finish my third part of my dissertation.

I have been very fortunate to work with many colleagues, Hung-Ching Chang, Shuaiwen Song, Bo Li, Matthew Grove, Hari Pyla, Aleksandr Khasymski, Sergio Bernales, Timmy Meyer, Kelsey Farenholtz, Min Li and Guanying Wang while at Virginia Tech. They helped me a lot when I was in Blacksburg.

I would like to thank the Department of Computer Science at Virginia Tech for providing such an excellent research environment for me.

Lastly, this work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. LLNL-TH-665255. In addition, this dissertation is based upon work supported by the National Science Foundation under Grant No. 0910784 and 0905187.

This page intentionally left blank.

TABLE OF CONTENTS

This page intentionally left blank.

CHART INDEX

This page intentionally left blank.

TABLE INDEX

This page intentionally left blank.

Chapter 1

Introduction

As of 2004, the microprocessor moved to multicore scaling—increasing the number of cores per die each generation—as its primary strategy for improved performance. Multicore processors can achieve higher computing throughput with adequate power consumption [22, 74], although the frequency of a multicore processor may be lower than that of a serial execution processor. Many in the microprocessor industry believe that this exponential multicore scaling will continue into the hundreds or thousands of cores within a single chip. For example, Intel introduced 48-core MIC architecture [8], the Cray XMT [143] system used 128 lightweight stream cores in a single chip, and the Tilera TilePro64 [10] system built 64-core on chip with a mesh on-chip network.

These multicore processors usually equip multiple memory subsystems to improve the data throughput. Non-uniform memory access (NUMA) is a typical multi-memory design used in these systems, where the memory access latency depends on the memory location relative to the core/processor. Under NUMA, a processor can access its local memory faster than non-local memory.

More recently, many in the research community and industry have discussed the introduction of heterogeneity to processors and memory systems to improve computation throughput and energy efficiency. Computing systems have started to include other heterogeneous computing accelerators for special-purpose tasks. The most dominant is the graphics processing unit (GPU), which was first intended to carry out graphics computations in parallel. Over time, GPUs have become more general, allowing them to be applied to general-purpose tasks with remarkable power efficiency. The Intel MIC is another example of heterogeneous computing; its architecture utilizes a high degree of parallelism in smaller, lower-power lower-performance Intel processor cores. It communicates with the CPU through a high-speed PCI bus. The result is improved performance on highly parallel applications. In addition, people also explore the hybrid use of traditional DRAM and emerging Non-Volatile Memory technologies, like phase-change memory (PCM) [36], STT-RAM [5], and memristors [7] to improve performance scalability, capacity, and energy efficiency.

With the increasing number of hardware resources and greater system complexity due to heterogeneity, future systems need more intelligent ways to manage hardware resources. Early studies geared toward improving performance and energy efficiency in these heterogeneous multicore, multi-memory systems focused on tuning a single or a few primitive(s) in the system. The key limitation of past research was a lack of holistic methodologies and resource management approaches to manage the tradeoff between performance and energy consumption. In addition, the shift from simple, homogeneous systems to these heterogeneous, multicore, multi-memory systems requires in-depth understanding of efficient resource management for scalable execution, including new models to capture the tradeoff between performance and energy, smarter resource management strategies, and novel low-level performance/energy tuning primitives and runtime systems.

In this chapter, we discuss the background for the research conducted in this dissertation. In particular, Section 1.1 discusses the challenges and defines the problems that we attempt to address concerning heterogeneous, multicore, memory-memory systems. Section 1.2

discusses the research objectives. Section 1.3 summarizes the contributions that we make in this dissertation. Finally, Section 1.4 outlines the organization of the full dissertation.

1.1.Research Challenges for Heterogeneous, Multicore, Multi-Memory Systems

To exploit the increasing of the number of cores in multicore systems, applications, programming languages and operating systems, we need to deal with parallel execution for processor throughput gains. Parallel programming is challenging: Programming for performance and energy efficiency is still a dark art, since the tradeoffs between performance and energy are not well understood**.** The tradeoffs between energy efficiency and performance that were well investigated in relation to serial processors in the 1990s have become more difficult to analyze in the multicore era due to the exponential increase in hardware resources and the complexity of computing systems. The community has spent over two decades trying to make the execution more efficient through the use of sophisticated memory hierarchies in giga- and tera-scale systems by improving data locality, and thereby lowering average memory access and bandwidth in our programs. The community has learned that efficient programs leverage small, fast caches close to the executing threads. We have also learned that minimizing the number of data communications between threads and the latency of those transmissions typically improves performance and energy. These design principles that were developed in the last two decades must be adapted to the heterogeneous multicore, multi-memory execution paradigm. However, many questions on how to design multicore, heterogeneous, multi-memory systems are still open and widely debated. In this section, we discuss two challenges that are critical for high performance and energy-efficient execution in multicore, heterogeneous, multi-memory systems.

1.1.1. Energy-Aware Computing

Today, multicore processors are the fundamental elements for large-scale high-performance systems. One of the most significant challenges for designing highly scalable parallel applications for larger-scale systems is a growing gap between the need for performance and the limits of the power envelope [70, 173]. A DARPA-commissioned report recommends a power of 20 megawatts for exascale systems [32]. However, several of today's most powerful supercomputers, armed with multicore nodes on the TOP500 List [14], require close to 15 megawatts of peak power. The power wall is already being reached by current petaflops systems (e.g., China's Tianhe-2 (MilkyWay-2)'s power requirement is at 17.8 MW [14]). Increasing the scale of HPC systems to improve computing throughput leads to serious reliability concerns for such systems due to heat emissions caused by high power consumption.

The energy consumption of the main memory system has also been growing [92, 114] due to emerging big data and HPC applications, which require a large amount of main memory bandwidth and capacity. Nowadays, memory sub-systems account for up to 40% of system energy [92]. This has become a new challenge for memory management using traditional DRAM technologies due to the limitations of high static power. Many people in the research community have begun to discuss the introduction of heterogeneity to the main memory to address this problem. They explore the hybrid use of traditional DRAM and emerging NVRAM technologies to improve performance scalability, capacity, and energy efficiency in the main memory systems.

To alleviate the power/energy crisis, the US Department of Energy has challenged the research community to build a supercomputer capable of exascale computations using less than 20 MW of power by the year 2022 [187]. To achieve this goal, the new systems must achieve a 1,000-fold performance improvement over current petaflops systems with only a

10-fold power budget [6].

Researchers have proposed novel software and hardware techniques to decrease power and energy consumption. The hardware optimization techniques include power capping [115], turbo/hyper-threading power states [41], and power configuration and power management in processor and memory sub-systems [22, 33, 58, 87]. These novel techniques drive new energy-efficiency achievements in the new computing system designs. On the other hand, software techniques provide several benefits for power/energy control schemes. First, they can leverage system-level information to aid decision making in their power/energy control schemes. For example, operation systems can provide a software stack to collect system-wide execution signatures, including processor and memory utilizations, frequency, and power states to assist the power/energy control schemes. They can also collect specific workload characteristics to optimize energy consumption. Second, software techniques provide more flexibility than hardware solutions. Software approaches can quickly adjust power control schemes, prediction models, and power budget plans to meet the needs of different system design requirements. In addition, a software approach can exploit the hardware control components, such as dynamic voltage and frequency scaling (DVFS) [100, 162, 200], thermal control [124], and power states [41] to control the energy consumption for computation.

Unfortunately, these existing energy-aware approaches are not directly applicable to emergent heterogeneous, multicore, multi-memory systems. Therefore, in this dissertation, we will adapt existing techniques and create new ones to improve the energy efficiency of emergent complex systems with increasing heterogeneity.

1.1.2. Resource Management

Efficient hardware resource utilization is the key for scalable execution. On a multicore system, concurrent thread execution needs to share hardware resources, such as the last level cache, high-speed bus, and off-chip memories, to improve utilization. Higher resource utilization improves collective performance, reduces the cost of hardware design, and even mitigates heat dissipation. However, resource sharing in multicore systems imposes new design challenges. In particular, more hardware resource sharing in concurrent execution can lead to performance variation and unwelcome energy waste. Resource management becomes more challenging in the multicore era due to the increasing number of cores and main memory devices (e.g., DIMMs, channels).

We use the following simple example to explain this problem. We ran the SP application from the NAS Parallel Benchmarks on an AMD 16-way, multicore, NUMA system. The SP application ran multiple times using 8 concurrently executing OpenMP threads with 85 different thread-to-core mappings. We analyzed 85 different mappings on 9 concurrent execution regions of the SP application. **Figure 1.1-1** shows the performance variance of the best, the worst, and the system default mapping. We found a performance difference between the best and the worst mapping of up to 45%. In addition, we found that the default system mapping did not guarantee the best performance. Compared to the default system mapping, the best mapping from the 85 selected mappings was 18% faster. However, the total number of mappings is $4.29*10^9$ (8 threads mapping to 16 cores). In this small example, we still have a large unexplored space to find the optimal solution, not to mention that it does not consider the data distribution in the memory system. Once the performance becomes unpredictable, violations of the system's performance requirements may occur.

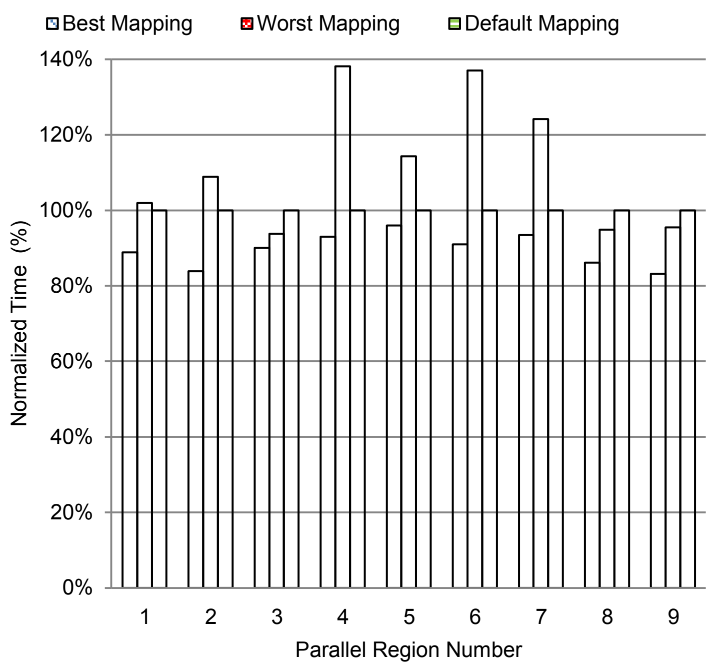


Figure 1.1-1 Performance variance of best, worst and default mapping among 85 mappings of the SP.A benchmark.

Another challenge arises in operation systems. Traditionally, operating systems have been responsible for managing shared hardware resources—processor(s), memory, and I/O. However, traditional operating systems lack mechanisms to detect hardware resource conflict due to concurrent execution. Hence, conventional operating system policies do not have adequate control over hardware resource management. To make matters worse, it is still not clear to the research community how multicore, multi-memory systems affect performance and energy. In terms of future applications, this is becoming a serious issue in the execution, as resource management mechanisms and policies are no longer adequate for future multicore, multi-memory systems.

Finally, it is not clear how heterogeneity will affect future resource management systems. While computing systems have started to include GPUs to accelerate computation tasks, how to distribute tasks between CPUs and GPUs has become a difficult design question for operating systems [189]. A task management system in the OS needs to have adequate tools to decide whether to off-load tasks from the CPU to the GPU, including explicit cost models and migration schemes. Today, most OSs do not have adequate tools to manage these tasks, creating a challenge for programmers. The same situation is emerging for heterogeneous memory systems. While big data and HPC applications drive the demand of the memory capacity, system designers must consider whether to add another non-volatile memory layer to traditional DRAM. However, it is still unclear to the research community how pages can be managed in heterogeneous memory systems that combine traditional DRAM and emerging NVRAM technologies to obtain optimal performance or energy [111].

1.2. Research Objectives

This dissertation aims to create a new energy-aware hardware resource management framework for future heterogeneous, multicore, multi-memory systems. It includes the development of efficient resource control schemes, building models that capture essential features of hardware resources, workload characteristics that affect performance and energy, and efficient management policies that will improve performance and energy.

The objectives of this research are as follows: (1) to develop an energy-aware, resource automation software framework that automatically identifies the optimal resource configuration based on hardware settings, workload characteristics, and execution signatures; (2) to identify, study, and build cost models for performance and energy problems that are related to resource sharing, contention, and throttling in multicore, multi-memory architectures; and (3) to build memory energy management strategies that extend the scalability for future heterogeneous memory systems and overcome the limitations of traditional DRAM technologies.

This dissertation consists of three parts. In the first part, we implement an automation framework to manage threads for performance and energy optimization. In the framework, we propose a memory-centric performance model to dynamically manage the number of threads used in a task. We also propose a thread-mapping algorithm to redistribute threads. In the second part, we propose energy-aware models that capture dominating factors of hardware resources that affect system performance and energy in multicore, multi-memory systems. Finally, in the third part, we propose new memory management policies in the memory controller that control the hybrid use of DRAM and emerging NVRAM technologies.

1.3.Research Contributions

In this subsection, we discuss the particular research contributions of this work. Each will be presented in more detail in subsequent chapters of this dissertation.

1.3.1. Improving Performance and Power Efficiency in NUMA Systems Using Thread Management

Non-uniform memory access (NUMA) is now the dominant memory system architecture for multiprocessors. NUMA has been a leading design paradigm in scalable, cache-coherent, multi-processor architectures since the 1990s.

Optimizing applications for performance and energy efficiency on NUMA architectures is increasingly challenging because more cores are being packed on each processor. While a significant body of prior work has treated NUMA as an issue of data distribution and migration assuming a stationary mapping of threads to cores [26, 133, 150, 151, 184], we consider the problem from the opposite direction: Given a distribution of data among memory nodes, what is the optimal mapping of threads to cores? In addition to the challenges of generating optimal static mapping of threads to cores, previous techniques to optimize power and performance dynamically on unified memory access (UMA) systems does not necessarily extend to NUMA systems. Earlier work [54, 116] has shown that *dynamic concurrency throttling* (DCT) is a viable optimization technique for performance and energy efficiency. DCT amounts to modifying (throttling) the number of threads to avoid oversubscribing hardware resources, such as shared memory bandwidth. DCT is beneficial when the degree of available algorithmic parallelism in a code region is less than the maximum number of cores available on the hardware. In a NUMA system, any attempt to throttle concurrency after execution begins will redistribute the computation between cores, thereby forcing extraneous cache misses, remote memory accesses, and contention. Prior work on DCT has overlooked this problem. In our work, we consider the optimization problem for multicore NUMA systems from the following three perspectives: (1) finding an optimal degree of concurrency, (2) mapping threads to cores to reduce remote accesses per core, and (3) minimizing contention on memory controllers.

We present DyNUMA, a framework for dynamic optimization of programs on multicore NUMA architectures. DyNUMA is implemented in a runtime system to improve both performance and energy efficiency. The core of DyNUMA is a novel memory-centric performance model. The model captures the nonlinear and interacting effects of concurrency, thread mapping, and data placement using a hardware–artificial neural network (ANN). DyNUMA uses an ANN model in conjunction with critical path analysis [11] to predict optimal concurrency and thread mapping, assuming static data placement.

This first part of the research makes the following contributions:

1. A flexible and portable framework, DyNUMA, to address the multidimensional problem of concurrency control and thread-to-core mapping on NUMA systems. DyNUMA dynamically controls the number of threads and thread mapping with minimal contention during the execution to optimize the performance and energy; and
2. A novel memory-centric, nonlinear performance model for NUMA architectures that captures the effects of concurrency, data placement, and memory contention on system performance. The model leverages the topology of the ANN to map the multicore NUMA architecture, and thus precisely captures the nonlinear performance effects of NUMA systems.

1.3.2. Modeling Performance and Power Efficiency in NUMA Memory Systems Using Queuing Methods

Ideally, as the number of cores and memory capacity increase, the consumption of additional power should result in a substantial increase in performance. In reality, an inadequate increase in hardware resources could have an adverse effect on performance and worsen energy efficiency.

In previous work, researchers focused on altering CPU and memory resources based on workload demand. These techniques include DCT [52, 116, 178], *memory throttling* (e.g., voltage/frequency scaling of DRAM) [58, 62, 214], and *memory parallelism control* [61, 130, 131, 213]. While these methods show promise in isolation, emergent systems must consider their combined interactive effects on energy efficiency. For example, our early research only focused on thread management using DCT and a thread-mapping scheme, without considering effects resulting from the memory system, such as the impact of the memory frequency and the number of memory nodes. To address this problem, in the second part of the dissertation, we propose an analytical model of memory performance that uses queuing theory to capture dominating factors of hardware resources that affect system performance and energy in multicore, multi-memory (NUMA) systems in a holistic fashion; these include thread-level parallelism (TLP), memory-level parallelism (MLP), and memory controller frequency. We use the resulting model to study the combined effects of DCT, memory throttling, and memory frequency on performance and energy, and address how to efficiently manage these hardware resources (i.e., threads, memory nodes, and memory frequency).

The second part of the research makes the following contributions:

- Our model predicts the application of CPI as a holistic function of TLP, MLP, and memory frequency to estimate the system performance. Furthermore, our models show that the memory frequency, MLP, and TLP have interacting effects on performance and energy. The effects of performance and energy cannot be considered in isolation; and

- We demonstrate that the model-guided optimization can improve energy consumption up to 40% for applications with high demand for memory bandwidth with proper control of resources including CPU cores, memory DIMMs, and memory frequency.

1.3.3. Managing Memory for Two-Level Heterogeneous Memory Systems

The memory wall has long been a computing bottleneck, and this has been intensified by the introduction of multicore processors. While the primary concern of the memory wall focuses on only bandwidth and latency, a new "power wall" challenge has emerged for scaling out memory capacity within a reasonable power budget. When big data and HPC applications drive the demand for memory capacity, traditional DRAM technology with high static power will unfortunately become less effective, and will not scale in terms of density and capacity.

Previous work [66, 108, 144, 158, 159, 196, 201, 209] has proposed that the power wall problem can be addressed using heterogeneous memories by exploiting DRAM for performance and emerging NVRAM memory technologies for capacity and energy efficiency. This work has proffered two basic policies to control the trade-off between delivering performance and improving energy consumption using two basic types of memory organization, namely PCache and HRank. PCache controls a hierarchical, inclusive system, while HRank controls a flat, exclusive system. We demonstrate that both PCache and HRank policies only exhibit good performance and energy for certain workloads.

In the third part of this dissertation, we propose a new memory controller (MC) design, namely HpMC, which employs the hybrid use of the PCache and HRank policies to manage memory pages and deliver optimal performance or energy based on system demand. Our research proposes a new memory controller design that manages memory resources (i.e., memory pages) in future heterogeneous memory systems.

The third part of this research makes the following contributions:

- We demonstrate via simulation that previous heterogeneous memory management policies exhibit good performance and energy only for certain workloads;
- We propose the first hybrid policies memory controller for heterogeneous memory systems, and our study demonstrates that better performance and energy can be achieved through the hybrid use of these policies via a well-designed MC; and
- The results show that the HpMC guarantees the delivery of optimal energy compared with its HM competitors, and improves energy consumption from 13% to 45%, while providing almost the same bandwidth as and larger capacity than the DRAM system.

1.4.Organization of the Dissertation

In Chapter 2, we discuss related work and present the background for the research conducted in the dissertation. Specifically, we present literature surveys for the following seven different topic areas: the performance models for multicore systems; DCT techniques and limitations for performance and energy optimization; performance models and optimization for memory subsystems; power modeling and optimization for memory subsystems; energy–aware management on multicore, multi-memory systems; data management for heterogeneous memories; and phase change memory optimization techniques.

In Chapter 3, we describe our research on improving performance and power efficiency for NUMA systems through thread management. We present an automation framework that adopts an ANN model and a thread-mapping algorithm to dynamically manage the number of threads and thread mapping.

In Chapter 4, we present novel analytical models for performance and energy efficiency for NUMA memory systems using queuing methods. The models consider important system factors such as TLP, MLP, and memory frequency. The models consider the combined interactive effects of these factors on system performance and energy, and overcome the limitation of previous works, where they only considered isolated effects. We investigate and evaluate the models on multicore NUMA platforms. We show the significant energy benefits brought from concurrency throttling, MLP throttling, and DFS.

In Chapter 5, we present a new memory controller design that combines the best aspects of two baseline heterogeneous memory management policies to optimize performance and energy. We validate our memory controller design in a simulation framework against real hardware on two state-of-the-art HPC servers. We investigate the effect of two policies on performance and energy using HPC workloads and analyze the effect of spatial and temporal locality on energy consumption in relation to both policies. Based on our locality analysis, we propose a new energy-aware hierarchical memory management policy that dynamically switches between the two policies to optimize energy.

Finally, in Chapter 6, we present a brief summary of the research done in this dissertation and future work to be carried out.

This page intentionally left blank.

Chapter 2

Background and Literature Survey

This chapter focuses on the background of the research and the literature survey on work attempting to improve the efficiency of parallel applications executed on heterogeneous, multicore, multi-memory systems. We will review numerous performance speedup models and summarize related power/energy-saving techniques and profiling methods. Since our models will approximate memory performance and power, we will also review memory performance and power models and related optimization techniques. We will review different energy-aware approaches on multicore, NUMA memory systems. Finally, we will discuss techniques to manage heterogeneous memory system and optimization approaches for phase-change memory.

2.1 Performance Models on Multicore Systems

As we enter the multicore and exascale era, we are at a pivotal point in the computing world. Computing vendors have designed chips with multiple processor cores. These upcoming chips are called chip multiprocessors, multicore chips, and many-core chips. Optimizing multicore performance will require further research in both extracting more parallelism and making sequential cores faster. For this reason, we need cost models that can capture the performance bottlenecks of multicore chips.

2.1.1 Analytical Models

Amdahl's law estimates the speedup of parallel design [24]. A number of researchers have proposed extended models based on Amdahl's law, such as Gustafson's law [82], Karp-Flatt metric [106], and models for multicore chips [30, 47, 48, 183, 203, 206]. Amdahl's law is based on the assumption of a fixed problem size. In contrast, Gustafson's law says that a larger workload can be solved within a fixed time when more parallel processors are given. The Karp-Flatt metric introduced the notion of load-balance and synchronization overhead, addressing the inadequacies of Amdahl's law and Gustafson's law. The metric can be used as a tool to measure the efficiency of the parallel execution of a given program.

Analytical models considering on-chip networking: Latency in networks-on-chips (NoCs) has become one of the critical factors in performance because more and more cores are being integrated into single chips. Xiaowen *et al.* [203] proposed a parallel speedup model extending from Amdahl's law. They considered the effects of network topology, network size, and traffic model, as well as the ratio of computation and communication. This analytical model can guide architects and programmers to improve the efficiency of parallel processing by reducing network latency and identifying the bottleneck.

Analytical models considering critical section and synchronization: Eyerman *et al.* [67] pointed out that parallel performance is not only limited by sequential codes, but is also fundamentally limited by synchronization through critical sections. They extended Amdahl's model to include critical sections, dividing critical sections into sequential and parallel parts. Their results showed that efforts to exclude critical sections can yield substantial speedup. Moreover, Chen *et al.* [43] discussed the impact of critical locks on performance in multicore systems. Their method identifies the critical sections appearing on the critical path, and quantifies the performance impact of critical locks on the critical path. Both of these studies tried to identify the synchronization overhead, which is not considered in Amdahl’s law.

Analytical models considering memory: Many applications cannot scale up to meet Amdahl’s law or Gustafson’s law due to memory constraints. Sun and Ni proposed the memory-bounded speedup model [181, 182], which is known as *Sun and Ni’s law*. This is a generalization of Amdahl’s law and Gustafson’s law. Moreover, Minjang *et al.* [109] estimated speedup by analyzing the cycles per instruction (CPI) on multicore systems. They approximated the on-chip and off-chip CPIs. Off-chip CPI is estimated by CPU memory stalls per last level cache miss.

Analytical models considering an asymmetric, heterogeneous design: There is also a body of researchers focusing on heterogeneous, asymmetric multiprocessors [30, 81, 91, 119, 135, 136, 165, 166, 190]. Hill *et al.* [91] applied Amdahl’s law to asymmetric multicores, concluding that asymmetric designs offer greater potential speedup than symmetric ones. However, the scheduling challenge needs to be well addressed in order to obtain a speedup. Meanwhile, Tong *et al.* [119, 190] explored the performance of asymmetric multicores using asymmetric schedulers. They used CPU clock modulation to quantify performance on an SMP and NUMA system.

Analytical models considering DVFS: Due to the widely used DVFS technique, some researchers have discussed the effects related to CPU frequency, performance, and energy [75, 100, 162, 200]. The researchers in [100, 200] assumed a per-core DVFS knob to be available and evaluated several different policies for a given power budget and performance estimation. Ge *et al.* [162], [88] developed analytical models to approximate performance and energy cost under different frequencies for scientific workloads on multicore systems.

2.1.2 Dynamic Approaches

Kismet [101] is a dynamic profiler that provides estimated speedups for a given serial program using binary instrumentation. It predicts speedup through critical path analysis by calculating self-parallelism for each parallel region. The overhead of this approach is significant. It shows 100x slowdowns due to the memory instructions instrumented. Meanwhile, Intel Parallel Advisor [9] collects timing information from an instrumented serial code and uses the information to build a dynamic parallel-region tree model to estimate the speedup.

Most existing works focus on the speedup with other factors, such as chip networks, critical sections, asymmetric task scheduling, and frequency scheduling, without considering memory effects. Some papers [101, 109, 181, 182] considered memory behavior in their model, but only assumed UMA memory systems, without considering NUMA effects.

2.2 Dynamic Concurrency Throttling

With the popularity of multicore architecture, many researchers developed concurrency throttling techniques to optimize multithreaded codes on multiprocessor systems. Voss *et al.* [195] carried out one of the earliest efforts to examine parallelism on shared memory multiprocessors. They proposed the notion of adaptive serialization, which takes critical sections and synchronization of parallel regions into consideration. They compared the measured parallel loop time and the predicted serial time to see if it would be useful for parallelization. The research only focused on using either one thread or a maximal number of threads, without considering the possibility of concurrency in between. Furthermore, Zhang *et al.* [211] proposed a self-tuning OpenMP loop scheduler designed to react to the behavior caused by inter-thread data locality. Zhang *et al.* [212] used a hardware-counter approach. Their scheduler samples the performance events directly from parallel loops and uses an off-line decision tree to decide how to schedule the loop to achieve load balance. These authors also predicted the best number of threads of a given parallel region instead of predicting performance. In addition, Suleman *et al.* [180] proposed a *feedback-driven* *threading (FDT)* framework to dynamically control the number of threads using runtime information. This system predicts the optimal number of threads by capturing the amount of data synchronization at execution time to mitigate bus saturation. The work took two factors into consideration, namely the amount of data synchronization and the bus bandwidth, but the researchers did not consider the cache contention on shared memory multiprocessors. Finally, Li *et al.* [118] mixed concurrency throttling and DVFS techniques to cap power consumption while maintaining a certain level of performance. The work focused on searching the configuration space and conducting empirical searches to reduce the total number of executions needed to be adapted. However, the experiments were based on simulation and overlooked the impact of overhead when changing from one configuration to another.

There are other compiler-based approaches [84, 104] based on dynamic feedback. They automatically determine the optimal number of threads for each parallel loop in the application at run time. They use a threshold method to determine how many threads they need to use in each parallel region; however, this approach does not exploit any runtime information to make better decisions, and only returns suboptimal solutions.

Curtis-Maury *et al*. [55, 174] used a machine learning approach to identify the concurrency configurations of SMP multiprocessors. Specifically, they used ANNs to predict performance and energy consumption under different concurrencies. ANNs greatly reduce the time in the training phase, which decreases the burden on the end user.

Most existing works leverage concurrency throttling based on performance prediction based on compiler time and runtime information; however, these works do not take the underlying memory topology into consideration. We extend previous work on DCT by leveraging the CPU and memory architecture topology to predict more accurate performance.

2.3 Performance Models and Optimization Techniques on Memory Systems

There has been a great deal of research focused on identifying memory as a future performance bottleneck. In the early 1990s, researchers concluded that memory bandwidth will limit the scalability of future systems, and predicted that future machines are likely to be memory bandwidth–bound due to the speed gap between processors and memory. This gap is also called the "memory wall" [140]. We first review several bandwidth and latency models, and then look at different techniques to optimize memory performance at different levels of the hierarchy.

Bandwidth and latency: Molka *et al.* [145] analyzed the memory system performance of Intel Nehalem in detail. In later work, Hackenberg *et al.* [83] compared the performance of Intel Nehalem with AMD Shanghai. Their analysis was based on using micro-benchmarks to measure the latency and bandwidth between different locations in the memory subsystem while considering the impact of cache coherency. Yang *et al*. [205] studied the effect of cache blocking and thread placement on multicore shared memory systems. They quantified execution time, but did not consider cache coherency traffic. Mandal [131] modelled the memory bandwidth and memory access latency of commercially available systems as a function of memory concurrency. However, extending this model to multiple types of memory controllers is difficult because the different rates of requests can only be produced through proprietary manipulation of the on-chip memory controller and interconnects. Other benchmarks measure the memory bandwidth but disregard most architectural details; one example of this is the well-known STREAM benchmark [137].

Cache partition: Tam *et al*. [185] addressed cache contention through software-based cache partitioning, similarly to many other researchers [45, 121, 123, 210]. In this approach, the cache is divided among all applications running on CPU using the page coloring technique.

Each application has private cache lines that the physical memory can map only into the private portion of the cache. The size of the reserved portion of the cache is determined by the application's reused distance profile. The reused distance profile is very similar to the stack-distance profile, which is approximated online using hardware counters [185]. The principle of cache partitioning is to isolate workloads of applications that harm each other. This approach has two limitations, as follows: (1) it requires customization of the complicated virtual memory system on the OS; and (2) it requires additional copy operations when the size of the cache partition changes or is reallocated. Task scheduling is not subject to these drawbacks.

Task scheduling: There is a large body of research focused on cache-aware and NUMA-aware task schedulers [69, 79, 129, 153, 161, 167, 175, 219]. Symbiotic job scheduling [175] is a cache-aware method for co-scheduling threads on SMT machines that minimizes resource contention. This method uses a brute-force method by trying a large number of thread assignments, picking up the assignment that yields the best IPC. Majo *et al.* [129] proposed a NUMA-aware task scheduler by measuring LLC pressure and NUMA penalty. Their algorithm requires application parameters that must be obtained online, which prevents dynamic adjustments to improve performance. Moreover, Zhuravlev *et al.* [35, 219] argued that LLC misses are not the only factor causing performance degradation; rather, the memory controller and pre-fetch mechanism are also important. They proposed an online task scheduler, but they still used the LLC miss rate as a metric to measure the extent of local contention. McCudy *et al.* [139] argued that NUMA problems can be identified with the help of hardware counters that track remote memory references. These crossbar events can now be counted in modern AMD and Intel architectures. We find that LLC misses are not the only factor in performance degradation, so we use the memory request event mentioned by Blagodurov *et al.* [35] as the metric to capture NUMA performance degradation.

Page migration: Page migration techniques improve performance by moving data to achieve a tight coupling of processors with resources. Chandra *et al.*'s [40] work represents one of the earliest efforts [31, 132, 134, 149, 168, 193] to examine the effects of OS scheduling and page migration policies on the performance of cache-coherent shared-memory servers. Their automatic page migration approach leverages TLB miss information to determine whether to migrate. In addition, Terboven *et al.* [186] extended the NUMA page placement policy called next touch to migrate pages that are frequently accessed remotely. Ribeiro *et al.* [160] used data access patterns to guide memory placement on NUMA systems, while Nikolopoulos *et al.* [150, 151] proposed a series of user-level dynamic page migration approaches.

Thread migration: Broquedis *et al.* [71] introduced a runtime system to optimize thread-to-data affinity using a BubbleScheduler scheme. BubbleScheduler remaps threads by employing a capacity metric to identify the memory nodes with the largest concentration of thread data. Threads are then migrated remotely to the identified node to maximize data reuse and minimize data transfer costs. Although this approach considers affinity, the focus is on modeling and optimizing data movement with threads tightly coupled to data. Despite the focus on minimizing data movement, as threads and cores scale, the need to migrate and the amount of data to migrate increase substantially.

Data placement: To the best of our knowledge, Awasthi *et al.* [25] were the first researchers to consider the problem of data placement with multiple memory controllers. They estimated performance degradation caused by congestion in a single memory controller, and found that the attribute costs of queuing delay and hit rates decreased in DRAM row operations. Blagodurov *et al.* [34, 35] argued that a middleware of shared resources must be extended to NUMA systems. They generated a detailed evaluation model of shared resource contention in multicore systems. Although they considered NUMA factors, they do not account for issues related to fairness of the queuing system in their system.

Memory controllers: We turn the discussion now to memory controller (MC) optimization. There is a significant body of recent papers [21, 50, 51, 99, 112, 126, 127, 147, 148, 192, 198, 208] examining multiple MCs in multicore systems. Loh [127] proposed a design that takes advantage of rich inter-die bandwidth in a three-dimensional (3D) memory chip. The memory 3D chip takes implements multiple MCs on chip that can quickly access several banks of DRAM simultaneously. The TilePro64 processor [4] uses multiple MCs on a single chip; all MCs are available via a mesh on-chip network. TilePro64 is one of the first commodity processors to use four on-chip MCs. Abts *et al.* [21] implemented different MC placements on a single chip processor to minimize the on-chip network traffic and controllers' channel load. Moreover, Vantrease *et al.* [192] discussed the interaction between MCs and on-chip networks and proposed MC layout solutions to minimize network traffic. Some papers have discussed MC scheduler policies [147, 148, 208]. Kim *et al.* [208] proposed a memory-scheduling algorithm that improves system throughput while minimizing coordination among all MCs. Mutlu *et al.* [147, 148] considered schedulers on a single MC, observing that the prioritization of memory requests to carry out row operations can lead to long queuing delays for threads that are not intended to access open rows. They proposed a *Stall-Time Fair Memory scheduler* to distinguish two parties, namely those that need to access open rows and those that do not. They also proposed a batch scheduler based on age and services of requests to achieve fairness of access time.

Most existing works model and optimize NUMA memory performance through quantifying bandwidth and latency. However, bandwidth and latency only partially explain the performance of applications. To the best of our knowledge, only a few models and optimization techniques estimate inclusive performance by considering the number of cores, bandwidth, latency, and NUMA effects and none of these are directly applicable to the heterogeneous multi-core processor and multi-memory configurations we explore.

2.4 Power Models and Optimization Techniques on Memory Subsystems

2.4.1 Power Models

Today's system designer is concerned about power used by the main memory in the system. Whether it is calculating battery life for a portable application or determining the power supply for a server, an accurate power budget for memory is essential. Several power models regarding DRAM technology have been proposed. We discuss several major DRAM power models here. Micron Inc.'s model [12, 13] is used extensively; this model uses SDRAM datasheets and measured current and voltage values for an application's behavior, such as page hit rates, to estimate the power consumption of a specific application. The model also provides some basic tools [13, 19] to calculate the system power consumed by the DRAM. The weakness of this model is that it needs accurate scaling of active/background components. In addition, it supports a close-page policy by default which is overly pessimistic compared to the normal case.

Joo *et al.* [207] predicted power and energy based on energy coefficients and an SDRAM energy state machine that independently characterizes dynamic and static energy. They explored energy behavior of the memory systems by changing design parameters such as processor frequency, memory frequency, and cache configuration. Rambus Inc.'s DRAM model was proposed in 2010 [18]; this estimates power based on device-level SDRAM details and technology specifications using switching activity. The drawback of this model is that device details are only available from memory venders.

2.4.2 Power and Energy Optimization

Lowering DRAM frequency: Deng *et al.* [62] proposed MemScale, a scheme that applies DVFS to the memory controller and DFS to memory bus and DRAM chips to save power and energy. This approach makes use of the OS's memory policy to decide on DVFS/DFS modes based on current bandwidth needs. Moreover, David *et al.* [58] assessed memory DVFS techniques in a real system, emulating reduced memory frequency by changing timing registers and using an analytical model to compute power drop.

Low memory power states: A body of studies has focused on how to utilize low power modes of DRAM, namely Rambus memories. Lebeck *et al.* [110] made use of page allocation policies to assist the OS and to complement the hardware power management. Their approach increases the chances that DRAM chips can be put into low power modes. Diniz *et al.* [65] proposed several techniques to limit consumption by controlling the power states of memory devices, as a function of the load on the memory subsystem. Specifically, they used optimization Knapsack algorithms to compute the optimal configuration of power states for a given power budget. Fan *et al.* [68] developed an analytic model that approximates the idle time of DRAM chips using an exponential distribution, and validated the model against trace-driven simulations. The trace-driven simulator processes instructions and data address traces of applications. However, the model ignores memory bus contention and the open/closed state of row operations in DRAM banks.

Huang *et al.* [97] proposed a power-aware virtual memory system that can effectively manage the energy footprint of each process through virtual memory remapping. Li *et al.* [120] built a model to estimate performance loss of low power management and proposed performance-guaranteed low power management schemes for both memory and disks. Pandey *et al.* [194] explored the unique memory behavior of DMA accesses to aggregate DMA requests from different I/O buses together in order to maximize the memory low power duration.

Capping temperature: Recent research has proposed techniques to limit peak power consumption or manage temperature. Lin *et al.* [124] proposed dynamic thermal management

(DTM) schemes to improve performance under the given thermal envelope. They proposed two schemes coordinating multicore and memory and adopted the clock gating and DVFS techniques for processor cores when memory was to be over-heated. Lin *et al.* further addressed the weakness of their previous study, namely the neglect of the CPU's heat dissipation and its impact on DRAM memories [125].

Other approaches: Delaluz *et al.* [60] proposed a compiler-directed and hardware-assisted hybrid approach to exploit low power models in memory. Moreover, Zheng *et al.* [213] proposed the mini-rank scheme, which adds a small bridge chip to each DRAM DIMM to break the DRAM ranks into multiple smaller mini-ranks. This approach increases the granularity DRAM access so that it can reduce the number of devices in a single memory operation to save power and energy. Hur *et al.* [98] used a history-based memory scheduler to manage power and energy. When additional DRAM power reduction was needed, they used a throttling approach to suppress DRAM activities by delaying the issuance of memory operations.

Most of the existing memory power models assume that only one memory controller and one DRAM module exist in the system. Our memory power model is similar to that of Micron, but is extended to multiple memory controllers with multiple DRAM modules. This represents a natural extension of the NUMA topology. Our optimization approach is different from existing techniques: We consider the best number of memory nodes needed for specific applications and turn the unused memory nodes to low power states.

2.5 Energy-Aware Management on Multicore NUMA Memory Systems

Performance and power are critical design constraints in today's HPC systems. Reducing power consumption without affecting system performance is a challenge for the HPC community. There is a significant body of research focusing on middleware and runtime systems. In early works, researchers applied DVFS techniques to data centers, while power consumption became a critical issue for large commercial server farms [27, 33, 38, 57, 64, 93, 218]. We introduce an instrumentation-based and transparent approach below.

Instrumentation-based approach: To exploit energy savings, the system needs to identify the potential regions of codes that can reduce energy consumption. Several studies have used the instrumentation-based approach, which involves (1) source code instrumentation for performance profiling, and (2) deciding on an energy policy based on profiled information. Cameron *et al*. [37, 76] used PMPI to profile MPI communications and exploit CPU idling to save energy. Moreover, Hsu *et al.* [95] used binary instrumentation to profile and insert DVFS scheduling functions to improve energy consumption. Freeh *et al*. [72] used PMPI to time MPI calls and decide whether or not to apply DVFS. These approaches require manual instrumentation to detect the inefficient regions.

Transparent approach: Several automated techniques have been proposed that are transparent to system users. Hsu and Feng [94] proposed the β-adaption algorithm to automatically adapt the voltage and frequency for energy savings at runtime. The user can specify the maximum allowed performance slowdown, and the algorithm will schedule CPU frequencies and voltages in such a way that the actual performance slowdown does not exceed what has been specified. Ge *et al.* [163] proposed *CPU MISER*, a performance-directed runtime system for power-aware computing. *CPU MISER* supports system-wide, application-independent, fine-grain DVFS. Moreover, it identifies several types of inefficient phases, including memory accesses, I/O accesses, and system idle under power and performance constraints. The researchers propose an accurate DVFS performance-prediction model that allows users to specify acceptable performance loss.

Lim *et al.* [122] proposed a runtime scheduler that captures MPI calls to identify the communication regions in MPI programs. Wu *et al.* [202] made use of a dynamic-compiler-driven runtime to identify the memory-bounded regions for power saving. Besides, CPUSpeed [16] adjusts power and performance modes based on the past processor utilization history. Tolentino *et al.* [188] proposed *Memory MISER*, which consists of a middleware in the Linux kernel that manages memory at device level and a userspace daemon that monitors memory demand systemically to control devices and implement energy and performance-constrained policies.

Our approach is different from existing techniques in that it exploits the idle time of cores and memories by lowering power states. Our framework determines the resources (number of memory nodes, and cores) needed for specific applications according to the execution signature. Our framework also decides on the thread-to-data affiliation to minimize power and energy consumption.

2.6.Data Management on Heterogeneous Memories

A few previous studies [108, 144, 197] considered combining NAND Flash and DRAM in the main memory to reduce the main memory power consumption. More recent works [39, 44, 46, 73, 86, 111, 113, 158, 159, 209] have focused on using PCM to partially or completely replace DRAM because this has more promising performance characteristics than Flash. Since page migration is the key to energy conservation and performance for main memory systems, several studies have attempted to address this problem. In a related approach [97], the authors proposed an OS-controlled, power-aware virtual memory periodically to migrate pages based on the reference bits. Although previous OS-only approaches have been able to improve energy consumption, OS latency is still a major concern. Several hardware-controlled systems were studied in [66, 194]. Pandey *et al.* [194] exploited the access pattern in workloads by frequently clustering accessed pages in a small subset of the memory chips to improve locality and energy consumption. Dong *et al.* [66] implemented an address translation mechanism in the memory controller that can dynamically migrate data between on-package and off-package memories. Such works simply looked at performance and energy through a single management policy; our work is different in that we explore the design space of combined policies to find the optimal solution.

Two memory hierarchy designs have been discussed in previous work. The first [158], [66] organizes the first level of memory as the cache of the second level. The second level memory is managed by the OS while the first level of memory is managed by the MC. The second design [159, 209] manages both memory layers as a flat address space. The idea of this design is to keep hot pages (high utilization) in the first level of memory, while migrating cold pages (low utilization) to the second level of memory. The MC [159] implements a variation of the 16-LUR MQ algorithm to rank the hot and cold pages and migrate them periodically. The OS does not immediately see the page migrations; it only updates its mapping of virtual pages to physical frames periodically from the address table of the MC.

2.7.Phase Change Memory Optimization

Lee *et al.* [111] proposed a buffer organization approach to narrow the row buffer sets to mitigate high energy PCM writes and exploit locality to coalesce write operations to hide latency. They also proposed a partial-writes technique, which maintains a bookkeeper to track data modifications and dirty cache lines in the MC to reduce migration traffic. Qureshi *et al.* [154] proposed a PreSET technique that monitors the modification of cache lines. As soon as the cache line is dirty, the PCM system initiates a SET operation to the memory cells required by the dirty cache line prior to the write operation. Thus, the write operation for the dirty cache line only needs a shorter RESET operation. Moreover, Yang *et al.* [204] proposed a data-comparison write (DCW) approach. The DCW scheme performs a read operation before the write operation to identify the previously stored data in the selected PCM cell. The scheme then compares flipped bits between the stored data and overwrites the bits that change. Hay *et al.* [89] proposed several policies that schedule write operations to maximize concurrent writes in the bank level based on the chip power budget and flipped bits information. This approach can improve both performance and energy consumption.

2.8. Summary and Conclusions

This dissertation focuses on improving the efficiency of high-performance systems with a particular focus on memory. To place our work in context, in this survey, we explored several related topics including system efficiency techniques (dynamic concurrency throttling, NUMA optimizations, data placement), techniques that further insight and understanding (performance models, power and energy models), and techniques that explore and exploit emerging systems (phase change memories, heterogeneous memories).

We also described some of our early findings that indicate performance and power have complex interactions not captured by the current state of the art techniques. For example, the interactive effects of resource contention on power and performance in caches, memory, network and I/O lead to inefficiencies in resource management at runtime. We also found that heterogeneity in memory and elsewhere in the system will play an important role for scalable and energy-efficient execution.

Our literature survey also indicates that performance and power models play a significant role in the prevailing runtime optimization techniques for thread scheduling, frequency control, memory throttling and data migration policies. This previous work motivates and forms the basis for the new modeling techniques proposed in this dissertation. In particular, our key observation is that existing models lack the detail needed to capture the interactive effects of power and performance. Nonetheless, we show that these interactive effects cannot be ignored and the effects grow with heterogeneity.

Thus, in the next chapter, we describe our techniques to improve the performance and power efficiency of thread management in NUMA systems by improving our ability to analyze the tradeoffs (using advanced modeling techniques) at runtime. This work in turn motivates our exploration of new analytical models described in Chapter 4. In particular, we create analytical models that capture the interactive effects of power and performance more precisely than previous work. These models consider the interplay of several critical factors, including thread-level concurrency, memory-level concurrency, and memory frequency. In Chapter 5, we leverage our improved understanding of the relationship between power and performance and extend the concepts to heterogeneous memories. Specifically, we present a new memory controller design that combines the best aspects of two baseline heterogeneous memory management policies to manage page migrations efficiently and to optimize performance and energy.

This page intentionally left blank.

Chapter 3

NUMA Scheduling

3.1. Introduction

Non-Uniform Memory Access (NUMA) is now the dominant memory system architecture for multiprocessors. NUMA has been the leading design paradigm in scalable, cache-coherent, multi-processor architectures since the 1990s. On a typical NUMA system, each processor has a local memory node accessible over dedicated links, while remote memory nodes are accessible via interconnects and through network interfaces. The latency of accessing the local memory node is markedly lower than the latency of accessing a remote memory node. More recently, non-uniform memory access latency is also present between cores in the same socket. The processor uses multiple memory controllers to serve its cores, with each controller connected to a different memory node. NUMA is therefore becoming pronounced also within the boundaries of a single chip. For example, the Tilera TilePro64 processor has four memory nodes on the same die[4]. It implements a shared physical address space via a mesh interconnect between cores. When a core accesses the closest memory node, it incurs lower access latency than when accessing other memory nodes. Similar asymmetric access latencies also appear in the NVIDIA Fermi architecture [3].

NUMA improves system scalability by avoiding bottlenecks in the memory subsystem and by increasing the memory bandwidth available per core. With an increasing number of cores per processor, NUMA is becoming necessary for systems to scale. According to Top500 statistics, over 90% of Top500 supercomputers are based on NUMA nodes [14]. Optimizing applications for performance and energy efficiency on NUMA architecture has been and remains challenging. While a significant body of prior work has treated non-uniform memory access as one of data distribution and migration, assuming a stationary mapping of threads to cores [26, 133, 150, 151, 184], we consider the problem from the opposite direction: given a distribution of data among memory nodes, what is the optimal mapping of threads to cores? As remapping of threads to cores is orders of magnitude faster than remapping data to memories, such an approach is worth considering as a dynamic optimization strategy.

Application performance is highly sensitive to thread-to-core mapping. **Figure 1.1-1** in Chapter 1 shows an example that quantifies performance variance due to different thread-to-core mappings on a NUMA system. We use SP from the NAS Parallel Benchmarks (class A, OpenMP version), running with 8 threads on a single node with 4 AMD quad-core processors. We enumerate 85 different mappings for 9 parallel regions in the benchmark. We observe a performance difference between the best and the worst mapping up to 45%. Compared to the default system mapping (Linux 2.6.32), the best mapping is 18% faster. Therefore, to optimize the performance and energy efficiency of applications on NUMA systems, we must determine the best mapping. However, the search space to determine the best mapping can be very large.

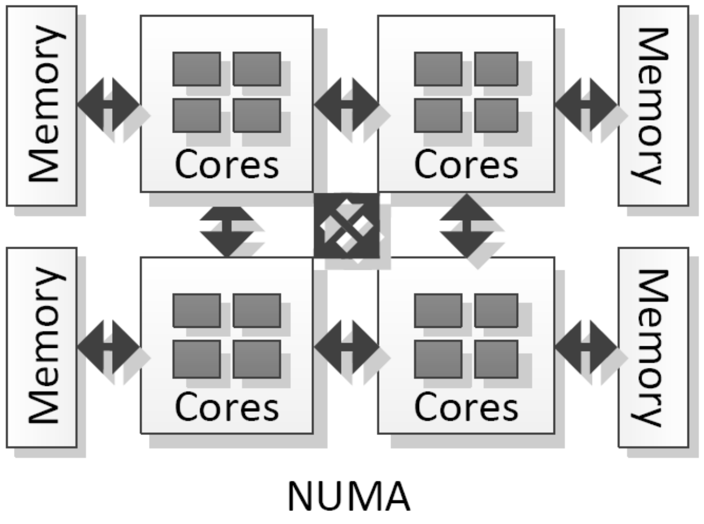


Figure 3.1-1 A 16-core NUMA architecture with 4 memory nodes

For the 16-core NUMA architecture shown in **Figure 3.1-1**, a system similar to the smallest system that we use in our experiments, there are over 63 million possible mappings of threads to cores, each with different memory access latency and bandwidth available per core. The above calculation excludes the impact of shared caches and assumes statically placed data. If we consider these implications, the search space is even larger.

In addition to the challenges of making optimal static mapping of threads to cores, previous techniques to optimize power and performance dynamically on Unified Memory Access (UMA) systems does not necessarily extend to NUMA systems. Earlier work [54, 116] shows that dynamic concurrency throttling (DCT) is a viable optimization technique for performance and energy efficiency. DCT amounts to modifying (throttling) the number of threads and the mapping of threads to cores used by parallel code at runtime, to avoid oversubscribing hardware resources, such as shared memory bandwidth. DCT is beneficial also when the degree of available algorithmic parallelism in a code region is less than the maximum number of cores available on the hardware. On a NUMA system, any attempt to throttle concurrency after execution begins will redistribute computation between cores, thereby forcing extraneous cache misses, remote memory accesses, and contention. Prior work on dynamic concurrency throttling overlooks this problem. In fact, any attempt to migrate threads or data in the operating system for the purposes of throughput, power optimization, or reliability, suffers from the same problem.

We consider a three-dimensional optimization problem for NUMA systems: (i) finding an optimal degree of concurrency, (ii) mapping threads to cores to reduce remote accesses per core, and (iii) minimizing contention on memory controllers. An optimal degree of concurrency avoids performance loss due to synchronization overhead, contention, or lack of sufficient algorithmic concurrency in the program. Reducing remote memory accesses reduces memory latency but may create contention due to oversubscribing of memory controllers.

Any solution to the optimization problem needs to identify the enumeration and layout of cores with respect to memory controllers and memory nodes (a non-trivial exercise) and also needs to consider phase behavior in programs such as changes in concurrency, memory access patterns or data communication and synchronization patterns [54]. Unfortunately, standard linear regression cannot capture the complexities of such systems. Non-linear regression models (or logistic regression) are often very complicated in formulation and can require substantial computation resources to solve.

To address these challenges, we created DyNUMA, a framework for dynamic optimization of programs on NUMA architectures through thread management. DyNUMA is implemented in the runtime system to improve both performance and energy efficiency. The core of DyNUMA is a novel memory-centric performance model. The model captures the non-linear and interactive effects of concurrency, thread mapping, and data placement using an Artificial Neural Network (ANN). ANN's are simpler to implement than logistical regression techniques requiring less formal statistical training. Furthermore, ANN's excel at deriving structure from data samples. DyNUMA uses an ANN model in conjunction with critical path analysis [179] to predict optimal concurrency and thread mapping, assuming static data placement.

3.2.System Design

DyNUMA optimizes OpenMP programs where parallelism is expressed with directives that delineate parallel regions. Each parallel region may enclose parallel loops, tasks, or nested regions. The design objective of DyNUMA is to select the best level of concurrency for each OpenMP parallel region and optimize thread placement to cores based on data locality so that the program is optimized for a given performance or energy-efficiency metric. The design of DyNUMA is based on the following characteristics:

- Scalable: system is expected to execute on architectures with massive parallelism.
- Architecture-aware: system should capture key architectural factors that affect performance and power.
- Light-weight: system should incur low overhead to allow for online dynamic optimization.
- Portable: system should be parameterized to allow for ease of porting to different NUMA architectures.

3.2.1. Overview

DyNUMA implements a dynamic online predictor for the degree of concurrency and the thread-to-core mapping of each parallel region. The framework is illustrated in **Figure 3.2-1**.

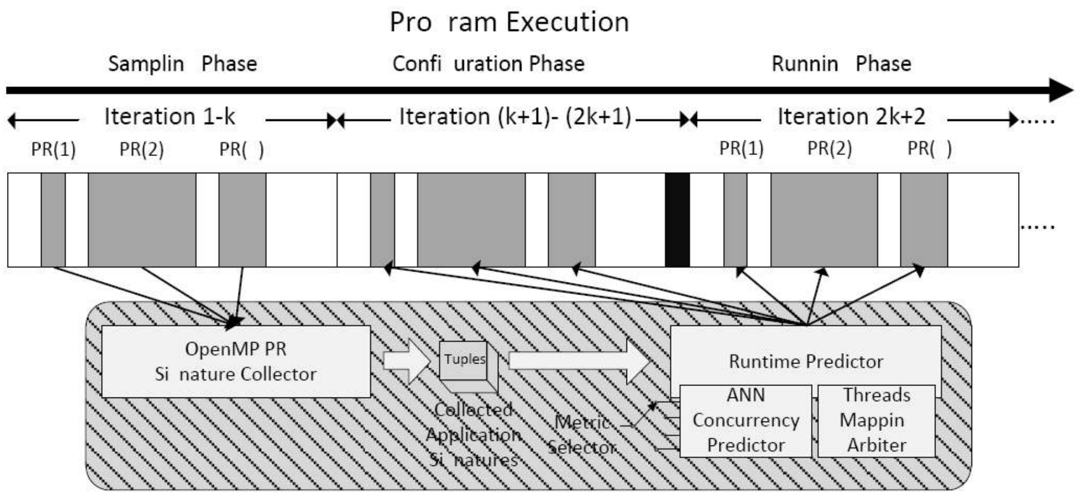


Figure 3.2-1 Diagram of the DyNUMA system framework.

The runtime predictor of DyNUMA includes two components. The first component is an architecture-aware, Artificial Neural Network Predictor (ANN) which predicts the degree of concurrency. The second component is a Thread Mapping Arbiter (TMA) which implements a deterministic algorithm that determines the thread-to-core mapping in linear time. DyNUMA assumes iterative programs where parallel regions are executed a number of time steps. This is common for many HPC applications. In the sampling phase, DyNUMA initially executes a program with maximal concurrency –using as many threads as the number of cores– for first k iterations. The number of k is equal to the number of memory nodes. The ith iteration samples threads' execution signatures on the memory node i. The choice of k is determined by a limitation of current hardware counters, that is, hardware counters can only profile one memory node at a time. Overcoming this limitation can significantly reduce k. DyNUMA samples all execution signatures during these k iterations to derive predictions of the best concurrency and thread mapping of each parallel region using ANN. Afterwards, DyNUMA applies TMA to further improve data locality. We define an execution signature as a collection of three metrics:

- *IPC*: Instructions per Cycle

- *LMA*: Local Memory Accesses per Cycle
- *RMA*: Remote Memory Accesses per Cycle

The runtime system collects the execution signature of each thread and transforms into a 3-element tuple. Each tuple characterizes a thread with respect to the intensity of computation to memory operations while executing a parallel region. *LMA* and *RMA* values are determined by the location of a thread. DyNUMA maintains *LMA* and *RMA* per memory node for each thread. DyNUMA uses thread-level tuples coupled with thread mapping information and observed metrics as inputs to the two DyNUMA predictors – ANN and TMA. *IPC*, *LMA* and *RMA* from all threads are used in the ANN to navigate the search space and predict performance on all degrees of concurrency. If an application is processor-bound, *IPC* should be high while *LMA* and *RMA* should be low. In this case, the ANN tends to select higher concurrency. Conversely, a memory bound application is expected to have low *IPC* and high *LMA* and *RMA* values, in which case the ANN tends to select lower concurrency to avoid oversubscribing the memory system. The optimal degree of concurrency can vary across regions due to variance of execution signature. On the other hand, TMA makes use of *LMA*, *RMA* and thread mapping information to redistribute threads in a more balanced way. Following prediction, DyNUMA actuates the selected concurrency and thread mapping for the remaining time of program execution.

3.2.2. Metric Selection

DyNUMA predicts performance and energy efficiency using the metrics shown in **Table 3.2-1**. The EDP and MFLOPS/Watt are calculated by **Equation 3.2-1** and **Equation** 3.2-2 respectively. The system provides the end user with flexibility to define different metrics while using the same unified prediction infrastructure explained in Section III.C.

Table 3.2-1 Three metrics used for the prediction of performance and energy efficiency.

Wall-clock time	Wall clock time of a parallel region
EDP	Energy-Delay-Product of a parallel region
MFLOPS/Watt	Number of floating point instructions (in millions) per second per Watt of a parallel region

Equation 3.2-1: $EDP = Power * (wall\ clock\ time)^2$

Equation 3.2-2: $MFLOPS/Watt\ = \frac{number\ of\ floating\ point\ instructions}{10^6 * Time * Power}$

3.2.3. Architecture-Aware Artificial Neural Network Predictor

One of DyNUMA's design goals is to be easily portable across platforms with different architectures. This is achieved by using portable metrics in the DyNUMA model of performance, namely *IPC*, *LMA* and *RMA*. The DyNUMA predictor uses a configurable, back-propagation, artificial neural network model [90] which can be ported by changing two parameters: the number of cores and the number of NUMA memory nodes of the target machine. ANN is an adaptive system that learns its coefficients using training sets fed through the network during a learning phase.

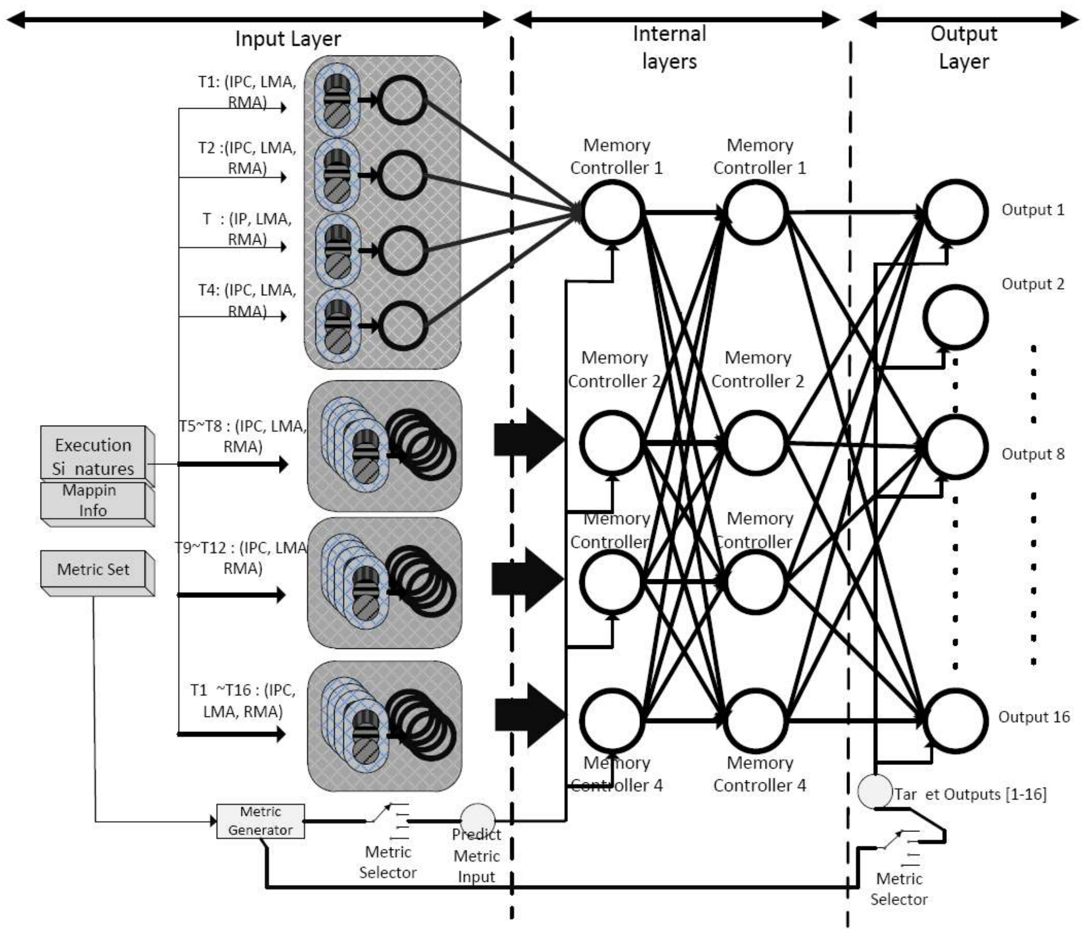


Figure 3.2-2 The ANN model for four quad-core processors (16 cores in total) and 4 NUMA memory nodes

Figure 3.2-2 shows an example of the configurable ANN model. The topology of the ANN model in this example emulates a node with 4 quad-core processors and 4 NUMA memory nodes. The topology of the ANN model emulates the target architecture. The ANN includes three layers: input, internal and output. The cells in the input layer correspond to cores and receive as input the execution signature of each thread. The cells in two internal layers emulate the controllers of NUMA memory nodes. The links between two internal layers emulate communication among memory nodes. For example, the link between the memory controller 1 and the memory controller 3 emulates data transfers between cores attached to the memory node 1 and cores attached to the memory node 3. The ANN can have multiple outputs. Each output represents the predicted metric at a different degree of concurrency. *Output i* is the predicted value when running the examined code region with i threads.

In the current implementation, the ANN model predicts the three metrics listed in **Table 3.2-1**. The ANN model can be reconfigured for different systems by changing the number of cells in the input and internal layers to correspond to different numbers of cores and memory nodes. The topology of the ANN reflects the system interconnect topology. It is not fully connected since each core is associated with one NUMA memory node and not all cores directly access all memory nodes. This ANN model can be easily adapted to handle SMT architectures (multiple hardware threads per core) by using the execution signature of hardware thread as input. There are several advantages of using ANN. First, it can easily capture the hardware architecture by changing its internal layers and topology. Second, it can generate multiple predictions under different levels of concurrency in parallel, contrary to prior linear DCT models that require a different model to predict each level of concurrency [54]. Third, ANN is a non-linear statistical modeling tool that captures complex relationships between inputs (execution signatures) and outputs (performance and power efficiency metrics). Such relationships cannot be easily captured by other models. We demonstrate this advantage by comparing the ANN model to a state-of- -art linear regression model proposed by Curtis-Maury et. al. [55].

1) Data collection: The ANN model in DyNUMA is trained offline. **Figure 3.2-1** shows the data collection framework for offline training. The OpenMP PR Signature Collector uses a set of APIs for application instrumentation. The instrumentation enables the collection of signatures of parallel regions, thread mapping information and metrics targeted for optimization. The signature collector uses PAPI[146], Oprofile [17] and WattsUp [20] power meters. The collected data is transformed into training samples. A training sample consists of: (1) a set of metric values (wall-clock time, EDP or MFLOPS/Watt), (2) a set of thread signature tuples, and (3) thread mapping information. *LMA* and *RMA* are hardware events and are collected using architecture-specific counters. The thread mapping data is collected with the portable *POSIX sched getcpu()* interface.

2) Power Measurement: To compute energy efficiency metrics (EDP and MFLOPS/Watt), DyNUMA collects power consumption of each parallel region. The runtime system uses an API to connect to external WattsUp power meters and record power for each region. The dynamic power of the two components varies as DyNUMA changes the number of active threads, memory access rate, and access pattern per thread. There are other hardware components that might exhibit dynamic power variance under DyNUMA; however, their power variance is expected to be relatively small, comparing to the processors and main memory [77]. We use **Equation 3.2-3** to compute the power variance of processors and memory:

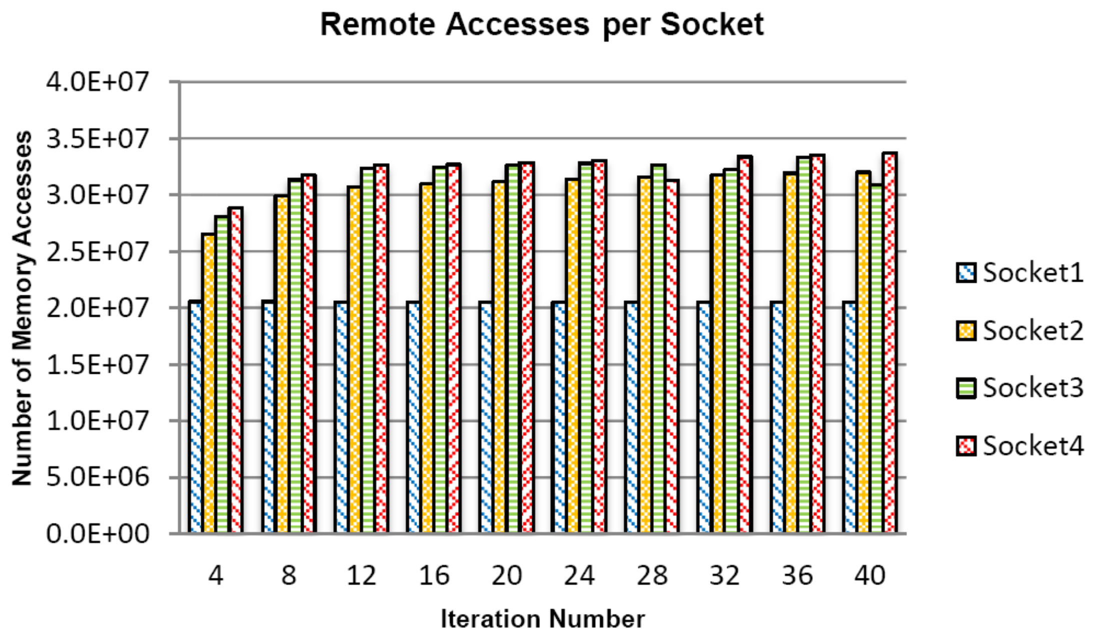
Equation 3.2-3: $Power = Power_{exec} - Power_{system\_idle}$

$Power_{exec}$ and $Power_{system\_idle}$ are collected from the WattsUp power meter. Because we are unable to physically access the TilePro64 machine that we use in our experimental analysis, power consumption of the TilePro64 processor is obtained from the TilePro64 technical specification, assuming that processor power scales linearly from idle (17 Watt) to maximum (23 Watt), with the number of cores.

3.2.4. Thread Mapping Arbiter

TMA uses an algorithm based on the critical path analysis to identify the optimal thread mapping. In most cases, programmers want to distribute workload (computation) evenly in their parallel execution. However, the execution time from one thread to another may still vary. This is because of different memory access patterns across threads and uneven distribution of data across memory nodes. The thread with the longest execution time in any given parallel region is said to be on the critical path. Note that TMA cannot be combined with ANN and has to be applied after ANN, because the critical path analysis can only be performed after the thread concurrency is determined.

We use **Figure 3.2-3** to further explain the critical path problem. **Figure 3.2-3** displays remote and local memory accesses per socket collected from the first OpenMP parallel region in the NAS FT benchmark (class B). The test was deployed on a platform with four quad-core processors (16 cores total), each with one memory node. We used 8 threads to run this parallel region and all threads are evenly distributed to 4 sockets (i.e., 2 threads per socket). We traced *LMA* and *RMA* per socket for 40 iterations. From the figure, we observe that each socket has different *RMA* and *LMA*. Socket 1 attains the lowest *RMA* and the highest *LMA*. We further mapped threads to cores in different ways, but a similar distribution of memory accesses was observed. The difference in the number of memory accesses results in asymmetric execution time between threads and causes the critical path problem.



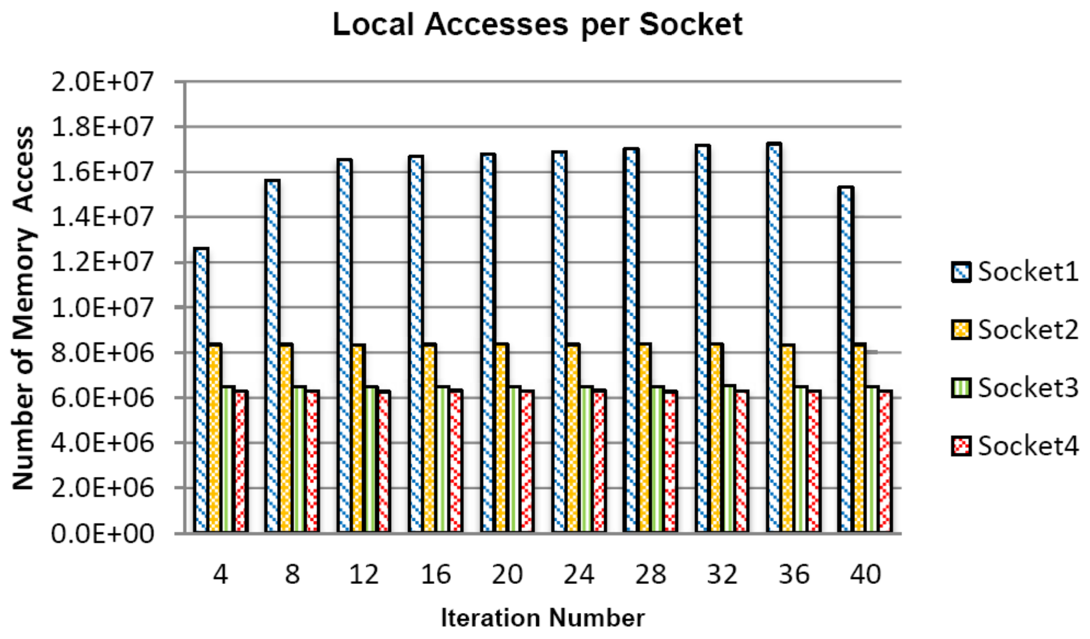


Figure 3.2-3 The distribution of remote memory accesses and local memory accesses in an OpenMP parallel region in FT.B

We present an algorithm that attempts to reduce the critical path by modifying thread placement, hence the ratio of local to remote memory accesses from each thread. The algorithm attempts to evenly distribute accesses between memory nodes, reduce remote memory accesses, and avoid contention on any memory node. The pseudo-code is shown in

Algorithm 3.2-1.

```
Algorithm 1 Thread Mapping Arbiter Algorithm
Input: TNT $t$
Output: Map $mapMinCp$
 1: Map $mapMinCp$ = $\Phi$
 2: Int cpImpact[Nd]= 0;  //The critical path impact on $Nd$
    //memory nodes
 3: ElementList sl= SortElementInTable($t$);
 4: while $sizeof(mapMinCp) \neq \#threads$ do
 5:     Element $e(T_i, D_j)$=GetMinCritcalPathElement(sl);
 6:     $mapMinCp$.Add($e(T_i, D_j)$);
 7: end while
 8: Return $mapMinCp$

 9: GetMinCritcalPathElement($ElementList\ x$)
Input: ElementList $x$  // A list of the candidate elements
Output: Element $e_{decide}$  // An element with the smallest
    //impact to the critical path
10: Element $e_{max}$ = GetFirstMaxElement($x$);
11: ElementList lc=FindAllPossibleCandidateElements($e_{max}$)
12: $e_{decide}$=FindLowestCPElement(cpImpact,lc);
13: RemoveThreadFromList($x$,$e_{decide}.T_i$);
14: AppendCriticalPathImpact($cpImpact$,IF($e_{decide}$));
15: Return $e_{decide}$

16: FindLowestCPElement(UINT64 $cpImpact[], List\ x$)
Input: ElementList $x$  //A sorted element list
Output: Element $e_{decide}$  //An element with the minimal
    //impact to the critical path
17: $minVal$=$UINT64\_MAX$; element $e_{decide}$=$\Phi$;
18: for all Element e in x do
19:     if $(IF(e) + cpImpact[e.D_j]) < minVal$ then
20:         $minVal$=$IF(e) + cpImpact[e.D_j]$);$e_{decide} = e$;
21:     end if
22: end for
23: Return $e_{decide}$
```

Algorithm 3.2-1 TMA Algorithm

The input to the algorithm is a thread to node mapping table (TNT). The output is the predicted best thread mapping (*mapMinCp*). The TNT is a data structure collects the number of memory accesses from each thread to each memory node, derived from the execution signature of the program collected during sample iterations. An example of a TNT is shown in **Table 3.2-2**. This TNT records the number of memory accesses from four threads to four memory nodes. Each element ($e(T_i,D_j)$), corresponds to the number of memory accesses to memory node j (i.e., D_j) from thread i (i.e., T_i). The algorithm first sorts all elements in the TNT in descending order of number of memory accesses (line 3 of Algorithm 1). This sorting step facilitates quick thread mapping in later steps of the algorithm. In the implementation, we use parallel radix sort to reduce sorting complexity. The sorting result is saved in a list (*sl*). Following the sorting, the algorithm iteratively selects an element from *sl* and places the selected element, $e(T_i,D_j)$, in *mapMinCp* (line 6) until all threads are selected. The selected element represents a decision of placing thread T_i on memory node D_j.

The selection criterion is implemented in *GetMinCriticalPathElement* (line 9). Generally speaking, this function chooses an element whose corresponding thread placement introduces the minimum imbalance of memory accesses between memory nodes. The function initially selects the first element from the sorted list (line 10), and then considers elements in other memory nodes (line 11) whose number of memory accesses are close (within 75% in our cases) to that of the first element in the input sorted list. The reason the algorithm considers multiple candidates instead of choosing the first element is that the first candidate from the list may not necessarily avoid imbalance of memory accesses between memory nodes. In particular, the first candidate may have a significant imbalance between *LMA* and *RMA* which creates unbalanced memory accesses across memory nodes. To estimate how placing a thread *i* on memory node *j* affects the critical path, we define a metric *Impact Factor*, *IF*, as:

Equation 3.2-4: $IF(T_i, D_j) = LMA_{i,j} + \sum_{k=1,k=j}^{N} NUMA\ Factor_{i,k} * RMA_{i,k}$

The equation weighs the number of remote memory accesses by a NUMA Factor because a remote access has longer latency than a local access. The NUMA Factor is the ratio of the remote memory access latency to the local memory access latency. The NUMA Factor is a variable. Depending on the distance between the core that issues a memory access upon a cache miss and the memory node where the miss is served, the NUMA Factor can have different values. The NUMA Factor can be calculated by measuring average access time when running a micro-benchmark to vary data location between memory nodes. Based on the above equation, an element with a small IF means that this element introduces lowest-unbalanced memory accesses between memory nodes while avoiding remote memory accesses. We also define a counter (*cpImpact*) associated with each memory node that accumulates the *IF* value for each memory node whenever a thread mapping is determined (line 14). The counter helps us trace the distribution of memory accesses across memory nodes.

FindLowestCPElement (line 12) selects the best candidate. For all candidate elements (line 18), the algorithm first calculates *IF(e)* + *cpImpact[e.Dj],* which estimates the impact of the memory accesses of a specific thread to memory node D_j on the critical path. The algorithm selects the element with the minimal value (lines 19 and 20) to minimize memory load imbalance between nodes while avoiding remote memory accesses.

Table 3.2-2 A TNT for 4 threads whose data is distributed into 4 memory nodes

Thread Id	Mem Node1	Mem Node2	Mem Node3	Mem Node4
1	100	1000	0	2000
2	1300	200	3500	1300
3	220	5000	500	500
4	4500	3800	2000	1000

Table 3.2-3 An example to show how we choose the best element

element	IFvalue	cpImpact	IF + cpImpact
e(4,1)	*IF(e(4,1))=14700*	*cpImpact*[1]=0	14700
e(4,2)	*IF(e(4,2))=15050*	*cpImpact*[2]=6830	21880
e(2,3)	*IF(e(2,3))=7700*	*cpImpact*[3]=0	7700

We use an example to further illustrate the algorithm. We assume a system with four threads and four memory nodes, with a TNT as shown in **Table 3.2-2**. After applying the algorithm, elements *e(3, 2), e(2, 3), e(4, 1),* and *e(1, 4)* are considered, which means that threads 3, 2, 4, and 1 are placed on cores close to memory nodes 2, 3, 1, and 4 respectively. We use a specific case to explain the process of choosing the best mapping candidate. In the second iteration of the selection loop (line 4), the algorithm first selects *e(4, 1)* from the sorted list. The algorithm selects this element, because it wants to first handle the element with the highest number of memory accesses. The selection of this element is the key to improve performance and should take the most favorable mapping when possible. However, *e(4, 1)* is not necessarily the best choice because it does not have the lowest *IF* on the critical path. Hence the algorithm consider other candidates (i.e., *e(4, 2)* and *e(2, 3)*). Their number of memory accesses is close to *e(4, 1).* The algorithm then calculates the IF values of the three candidates and checks the *cpImpact[j]* on each memory node (shown in **Table 3.2-3**). The algorithm eventually selects *e(2, 3)* instead of *e(4, 1)* because its *IF* + *cpImpact[j]* is the lowest among the three candidates, which intuitively introduces the smallest imbalance between the four memory nodes.

3.2.5. Overhead and Penalty Control

DyNUMA changes concurrency and thread mapping between parallel regions. Frequent changes in concurrency may incur performance loss due to cache flushing. To ameliorate this effect, the runtime system considers remapping threads only for parallel code regions with sequential execution times of 100 milliseconds or higher. In addition to cache flushing, non-optimal concurrency prediction or non-optimal prediction of thread mapping can cause performance loss. DyNUMA uses an additional iteration to measure performance of the selected configuration and compares it with the performance of the system default. If the system default is better, the predicted configuration is discarded, and the system default is taken.

3.3.Performance Evaluation

Experimental analysis explores two aspects of DyNUMA: prediction accuracy of the ANN model and effectiveness of model-based optimization. We use two benchmark suites, the NAS parallel benchmarks (3.1) [28] and the ASCI Sequoia benchmark suite [177]. The benchmarks have 85 OpenMP parallel regions in total. Their workload ranges from compute-intensive to memory-intensive and most benchmarks exhibit phase changes in their memory access patterns. We use the Class D data set for all NAS benchmarks and use two of the Sequoia AMG benchmarks, AMG.Relax and AMG.Matvec. The number of sample iterations *k* (see Section 3.2.1) is 4 in our tests. When presenting the results, we use the notation ***benchmark_suite_name.benchmark_name.region_no*** to represent a specific OpenMP parallel region. For example, NPB.FT.1 refers to the first parallel region in the benchmark FT in the NAS benchmark suite. We present experiments from three platforms listed in **Table 3.3-1** to verify the portability of DyNUMA. We use Intel's C and Fortran compilers (version 12.0.2) on AMD platforms. On TilePro64, we use the Tilera GCC and Fortran compiler (version 3.0.1) to perform cross compilation on an X86-64 platform.

Table 3.3-1 Three test platforms

Processor	#Cores	Speed	Memory Nodes	Memory Size
Barcelona	16	2.0 GHz	4	64GB
Magny-Cours	32	2.5 GHz	4	128GB
TilePro64	64	866 MHZ	4	64GB

We execute OpenMP benchmarks with static loop scheduling, which is the most appropriate for the selected benchmarks. Nevertheless, DyNUMA is independent of scheduling policy and can be applied as is once an initial distribution of workload between threads is performed by the scheduler. We execute benchmarks using first-touch for data placement in memories. First-touch is a page-level placement policy that allocates each page in memory located as close as possible to the processor that first touches the page during program execution. First-touch is an effective common case policy for many operating systems (e.g., Linux and FreeBSD).

3.3.1. ANN Model Prediction Accuracy

We evaluate the ANN model prediction accuracy by predicting wall-clock time and EDP. We use a cross validation technique in our experiments. In particular, we use 7 out of the 8 benchmarks for training and the remaining benchmark to verify prediction accuracy. **Figure 3.3-1** shows the prediction error rate on the three platforms using 1400 samples in total. The error rate for wall-clock time is 2.18% on average and only 7.7% of the samples has an error rate higher than 5%. The prediction error rate for EDP is 3.31% on average and only 13.9% of the samples has an error rate higher than 5%.

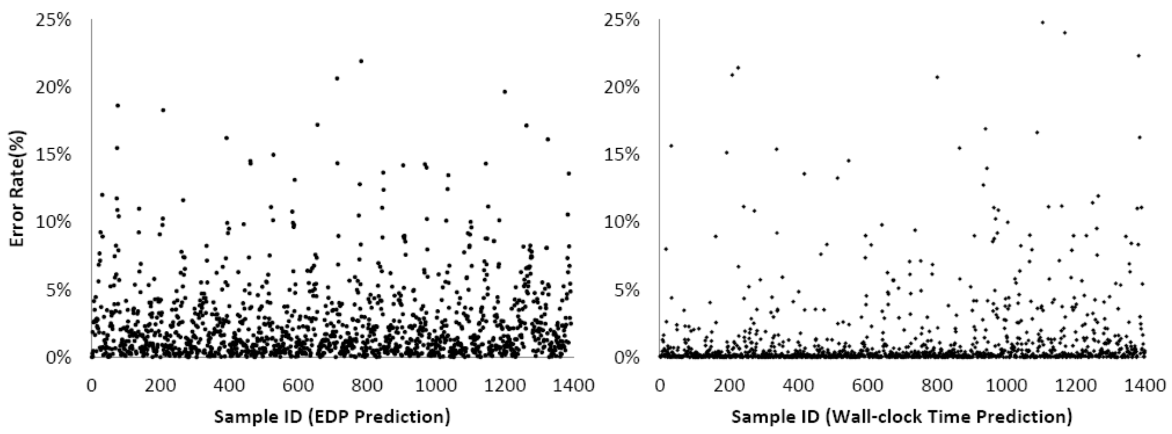


Figure 3.3-1 The distribution of ANN prediction error rate for EDP and wall-clock time

To investigate the variance of prediction accuracy across benchmarks, we look into the prediction results for each benchmark.

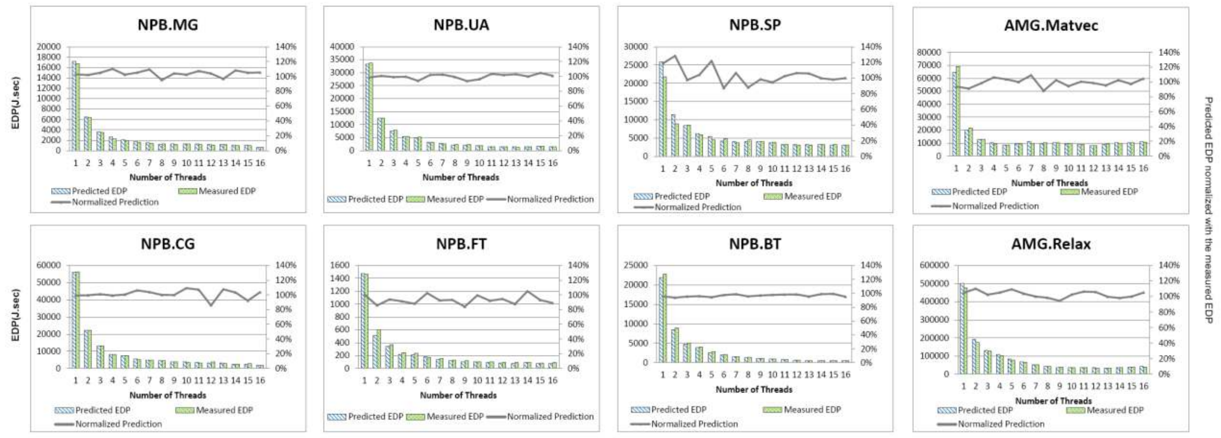


Figure 3.3-2 The EDP prediction results for the 16-cores system with the ANN model. The *Normalized Prediction* refers to the predicted value normalized by the measured one.

Figure 3.3-2 displays the EDP prediction results for one OpenMP region of each benchmark. Similar variance of prediction accuracy is observed in other OpenMP regions. We notice that the predictor achieves high accuracy no matter how many threads are chosen to run a parallel region. We also notice that the prediction error rate for NAS SP is relatively high. We suspect this is due to a shift in the memory access pattern within the benchmark region studied. Our model cannot capture well oscillating memory access patterns within the same

OpenMP region. Prediction accuracy can be improved, if the model is applied at a granularity finer than that of an OpenMP parallel region.

3.3.2. Comparison between ANN Model and Linear Regression Model

Linear regression models have been used for performance prediction in earlier work [54, 116, 117]. They are a realistic baseline to compare against the ANN model. We compare the prediction accuracy of the ANN model with that of a linear regression-based model proposed by Curtis-Maury et al. [53]. This linear regression model is briefly explained in **Equation 3.3-1**.

Equation 3.3-1 : $p_{i} = P_{max} * Hi(m1, m2, m3, m4) + e_i$

Here, p_i is the prediction target (e.g., wall-clock time, EDP or MFLOPS/Watt) for the case of using *i* threads. P_{max} is the measured value using maximal number of threads and $H_i()$ is a transfer function to scale the observed P_{max}. The transfer function is a linear combination of four hardware event rates, *m1,m2,m3,* and *m4,* with significant contribution to the observed metric, in a statistical sense. For the 16-core Barcelona system, these rates are *IPC*, *LMA*, *RMA* and branch misses per cycle. e_i is a constant residual.

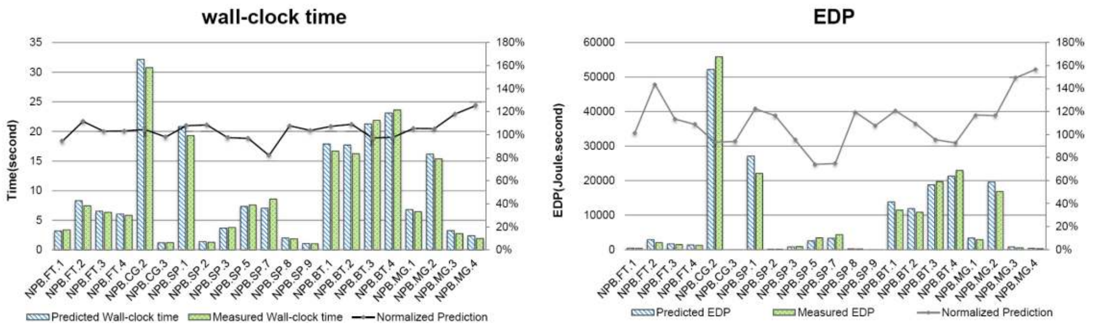


Figure 3.3-3 Prediction accuracy of the linear regression model

Figure 3.3-3 shows the prediction results from 21 parallel regions of NPB FT, CG, SP and MG benchmarks using the linear regression model. The benchmarks run with 8 threads on the 16-core Barcelona platform. The curves within the figure represent prediction values normalized to the measured values. We find that linear regression predicts EDP poorly.

Table 3.3-2 Comparison of the linear regression (LR) and ANN models for time and EDP predictions

Model	LR	ANN
The averaged error rate for time prediction	9.90%	2.18%
The standard deviation for time prediction	1.591	0.156
The averaged error rate for EDP prediction	22.61%	3.31%
The standard deviation for EDP prediction	2741.3	106.78

The prediction error is up to 60%. We further compare the linear regression model and ANN models in **Table 3.3-2**, which summarizes the prediction error rates for wall-clock time and EDP, collected from the 16-core Barcelona platform. The results are averages of 21 parallel regions. In terms of wall-clock time prediction, the ANN model is about 7% better than the linear model, with the standard deviation being 10 times less. In terms of EDP prediction, the ANN model is much better (18%) than the linear model, with the standard deviation being 25 times less. The ANN model achieves better prediction accuracy than the linear model. This is because there is inherent nonlinear relationship between hardware counter event rates and the prediction target, due to the implications of data locality and contention. The linear model lacks the ability to emulate the NUMA architecture, as all remote memory accesses are treated equally and summarized as a single term with only one coefficient within the model, despite varying latency due to the interconnect topology and contention. In contrast, the ANN model can map data locality and architecture details into the model illustrated in **Figure 3.2-2** , hence is able to make prediction with higher accuracy.

3.3.3. Thread Mapping

We compare thread mapping in DyNUMA to the default thread mapping scheme used in Linux. **Table 3.3-3** displays selected results. For each benchmark, we choose a specific number of threads and then execute it with the two methods to decide the thread mapping. We run each test 100 times on the 16-core Barcelona machine. **Table 3.3-3** reports the best performance improvement with DyNUMA for each test case. The results indicate that optimized thread mapping can significantly improve performance.

Table 3.3-3 Performance improvement with our thread mapping algorithm

Benchmark	# Threads	Performance Improvement
SP.C	4	20%
FT.B	8	28%
MG.B	12	6%
MG.B	16	14%

3.3.4. ANN versus TMA

We use two benchmarks, AMG.Relax and AMG.Matvec to show if the ANN predictor provides performance improvement over a system that uses only TMA as an optimizer before showing the performance of the two optimizers combined in next subsection. **Figure 3.3-4** shows that concurrency control with the ANN provides significant additional improvement in performance and energy-efficiency compared to mere thread mapping optimization. This behavior is more pronounced in memory-bound code regions.

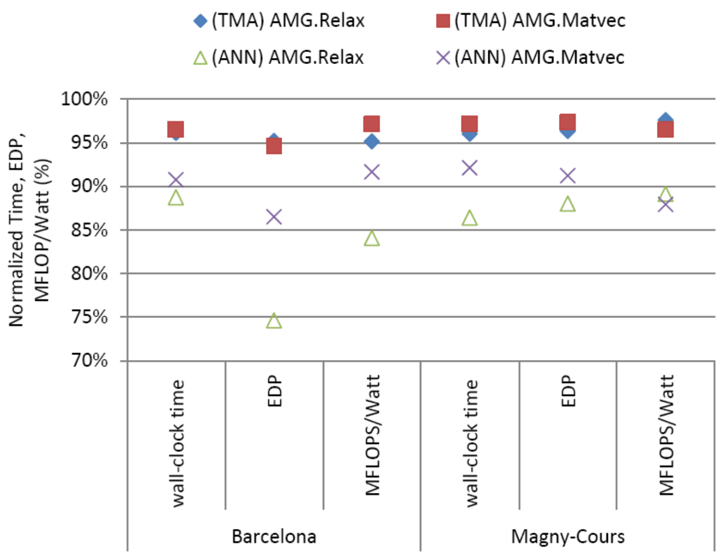


Figure 3.3-4 Performance comparison of ANN over TMA

3.3.5. DyNUMA Results

We report results in **Figure 3.3-5** to **Figure 3.3-8** and **Table 3.3-4**. These results are normalized to the respective metrics with maximum concurrency and the default Linux thread mapping.

On TilePro64, we test DyNUMA with a limited subset of the benchmarks due to hardware instability. The TilePro64 provides a platform-specific Oprofile tool for collecting hardware event rates. Oprofile, unlike PAPI, does not have the ability to collect data at runtime. Therefore, the TMA algorithm cannot collect application signatures on TilePro64. Hence, we only use the ANN model to predict thread concurrency without applying TMA on the Tilera platform. **Figure 3.3-5** summarizes the performance of DyNUMA and **Table 3.3-4** presents averages. We notice significant improvement in EDP and noticeable improvement in wall-clock time on the TilePro64. The improvement stems exclusively from concurrency throttling, as applications do not scale perfectly on the TilePro64. By choosing appropriate thread-level concurrency, DyNUMA improves EDP by 30%. Improvements in performance and energy-efficiency on other platforms are more modest but still measurable and consistent.

To further explore DyNUMA results, **Figure 3.3-6** to **Figure 3.3-8** break down the metrics presented in **Figure 3.3-5** between OpenMP parallel regions longer than 100 milliseconds. On the 16-core Barcelona system, DyNUMA achieves improvement in performance in 45% of the OpenMP parallel regions and energy efficiency in 72% of the OpenMP parallel regions; on the 32-core Magny-Cours machine, DyNUMA achieves improvement in performance in 59% and energy efficiency in 56% of OpenMP parallel regions; on the Tilera platform, all parallel regions benefit from DyNUMA in both performance and energy efficiency. However, not all parallel regions present opportunities for optimization. Compute-intensive regions tend to be more scalable and less sensitive to thread mappings than memory-bound regions. This is the case, for example, in NPB.FT.4, NPB.BT.1, NPB.BT.4 and NPB.UA.18. In these parallel regions, DyNUMA leads to negligible performance loss.

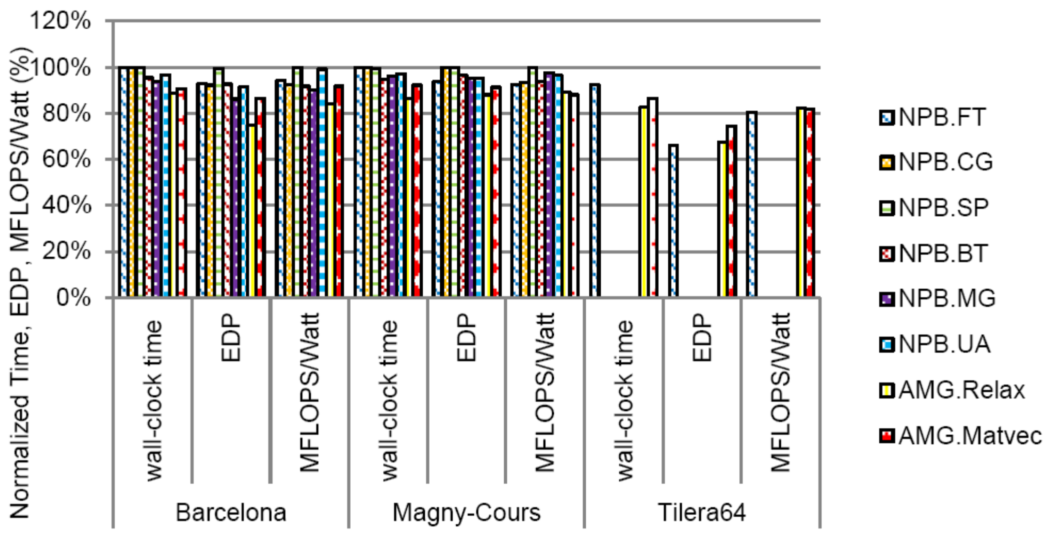


Figure 3.3-5 Performance improvement with DyNUMA on the three platforms.

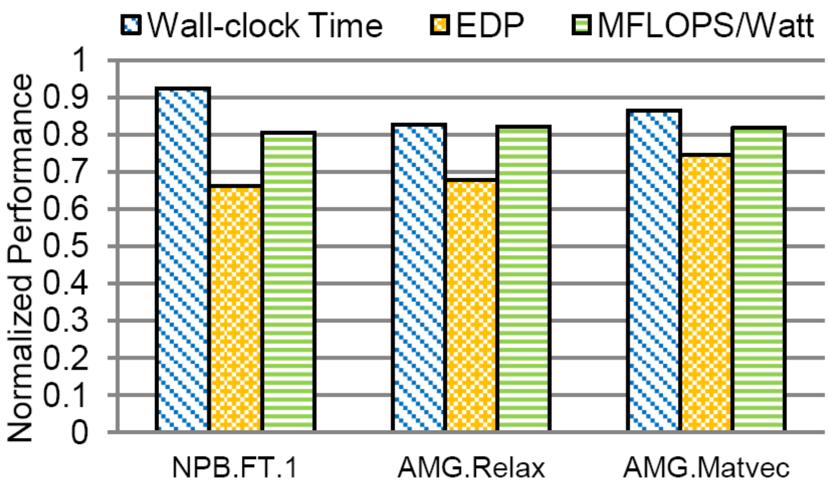


Figure 3.3-6 Performance with DyNUMA on the 64-cores Tilera platform

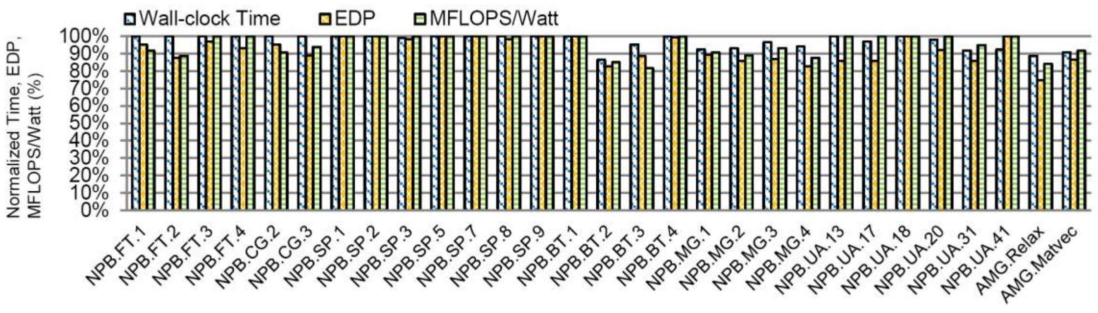


Figure 3.3-7 Performance with DyNUMA on the 16-cores Barcelona platform

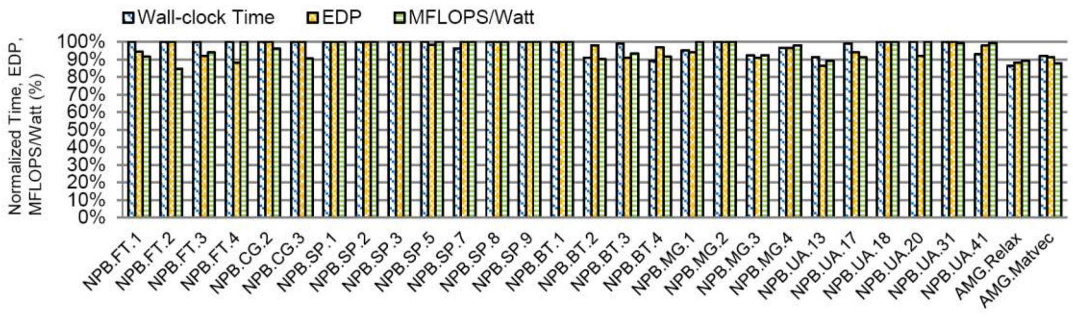


Figure 3.3-8 Performance with DyNUMA on the 32-cores Magny-Cours platform

Table 3.3-4 Performance improvement with DyNUMA on the three platforms

Metrics	Barcelona	Magny-Cours	TilePro64
wall-clock time	6.74%	6.58%	12.88%
EDP	10.45%	6.90%	30.58%
MFLOS/Watt	10.66%	7.60%	18.49%

3.4.Conclusions and Future Work

Performance and energy efficiency optimization depend on effective control and mapping of parallelism to the system architecture. NUMA architectures significantly expand the search space of optimality. Programmers are often unaware of or unwilling to navigate this space via experimentation. Effective automatic control of concurrency and mapping needs to consider not only workload characteristics but also specifics of the underlying NUMA architecture.

This work presents a framework combining a memory-centric, architecture-aware ANN model and a thread mapping arbiter to help parallel programs to autonomously optimize their concurrency and thread mapping at runtime.

We evaluate the framework using the NAS and Sequoia Benchmarks on three different NUMA platforms. DyNUMA achieves on average 8.7% improvement in wall-clock time, 16% improvement in EDP and 12.3% improvement in MFLOPS/Watt.

For future work, we will incorporate DyNUMA with dynamic data migration to achieve better thread-data affinity. We will also develop a strategy to combine small parallel regions into bigger ones to explore new opportunities for performance improvement.

This page intentionally left blank.

Chapter 4

NUMA Modeling

4.1.Introduction

Since the dawn of the multi-core era, demand for memory bandwidth resources has increased dramatically because of the rapid increase in the number of cores per chip. Transferring growing amounts of data between the CPU and memory at higher rates of speed generally increases power consumption while the performance gains vary from substantial to nonexistent due to the memory wall. Ideally, as the number of cores and memory capacity increase, consuming additional power should result in a substantial increase in performance.

In previous work, researchers focused on altering memory bandwidth dynamically based on workload demand while lowering power consumption. These techniques include dynamic concurrency throttling (e.g. thread/core control) [52, 116, 178], memory throttling (e.g. voltage/frequency scaling of DRAM) [58, 62, 214], and memory parallelism (e.g. memory node control) [61, 130, 131, 213]. Dynamic concurrency throttling not only controls the computation throughput but also controls the demand of bandwidth to the memory system. In addition, memory throttling (i.e. DVFS) and memory parallelism controls the theoretical maximal bandwidth supported in the memory system. While these methods show promise in isolation, emergent systems must consider their combined interactive effects on energy efficiency (i.e. power and memory bandwidth).

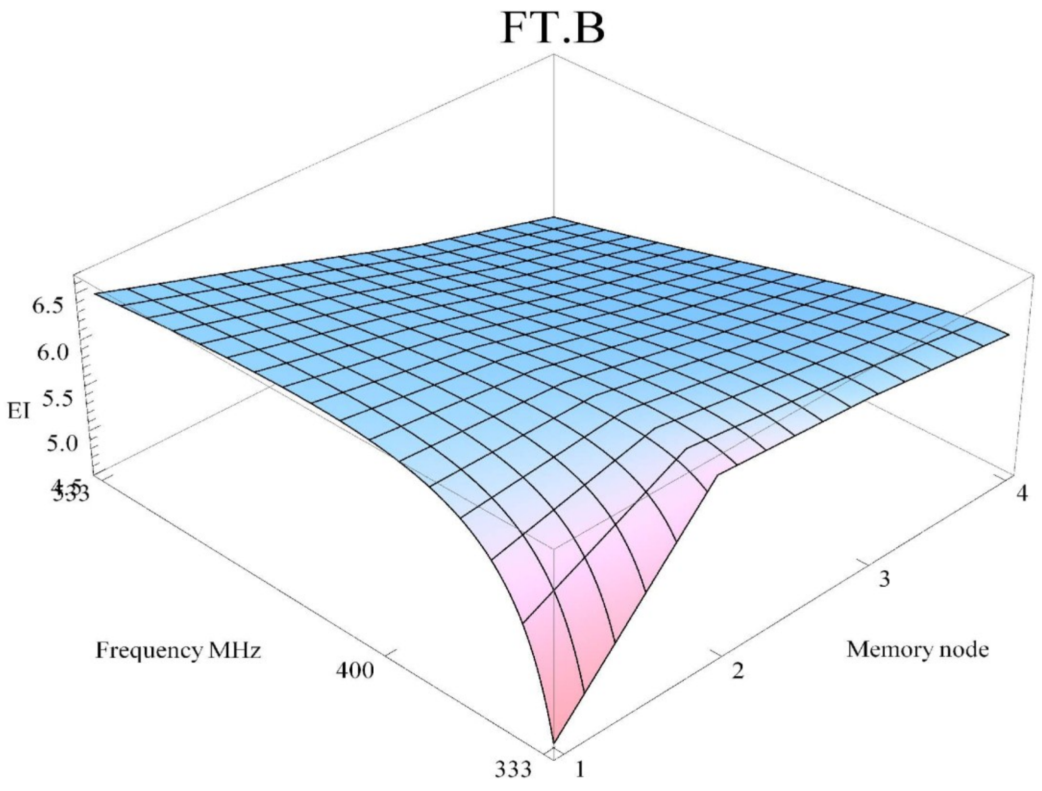


Figure 4.1-1 Energy improvement of FT benchmark

As shown in **Figure 4.1-1**, the energy consumed by FT on a 4-socket, 16-core system is a function of memory frequency and the memory-level parallelism (MLP). FT, part of the NAS parallel benchmarks (NPB), solves a 3-dimensional partial differential equation using the fast Fourier transform. The vertical axis represents the energy improvement (higher is better) over a baseline configuration of 1 memory node at 333 MHz frequency. For FT, using more than 2 memory nodes at 400 MHz frequency and higher provide diminishing improvements in energy. Finding the optimal energy is a multi-dimensional optimization problem because of the interacting effects of memory frequency and memory parallelism. Furthermore, although not shown in **Figure 4.1-1**, thread-level parallelism (TLP) is also an important parameter as it affects memory contention and overall performance. Understanding the combined effects of memory throttling, memory parallelism, and thread parallelism on performance and power is a challenging task. In this section, we propose an analytical model of memory performance that uses queuing theory to capture the effects of contention on bandwidth [78]. We use the resulting model to study the combined effects of dynamic concurrency throttling, memory throttling, and memory parallelism on performance. We demonstrate that model-guided optimization can improve energy consumption up to 40% for applications with high demand for memory bandwidth.

Our performance model, concurrency-frequency model or CFM, predicts the number of cycles per instruction (CPI) as a function of the number of threads, the number of memory nodes, and the memory bus frequency. This model is based on an M/M/C queuing model [78], which we describe in Section II. We use CFM to derive an energy model for multi-core, non-uniformed memory access (NUMA) systems based on power models from published vendor data [5, 23, 215]. Our energy model is described in Section III. We validate our models of performance and energy in Section IV. In Section V we analyze the impact of thread parallelism, memory frequency and concurrency, and their combined effect on energy and performance. Our related work is described in Section VI followed by our conclusions in Section VII.

4.2. A Preliminary Queuing Model

In this section, we first introduce the memory system in a NUMA multi-core multiprocessor. We then discuss how to apply a customized M/M/C system to model the memory system performance based on the queuing theory. This M/M/C system serves as our preliminary model to motivate our further work.

4.2.1. Memory System in NUMA Multi-core Multiprocessors

We use **Figure 4.2-1** to illustrate the memory system in NUMA multi-core multiprocessors. Most of modern multi-core multiprocessors have memory controllers (MC) integrated into processors, but we separate MC from processors within the figure for illustration purposes.

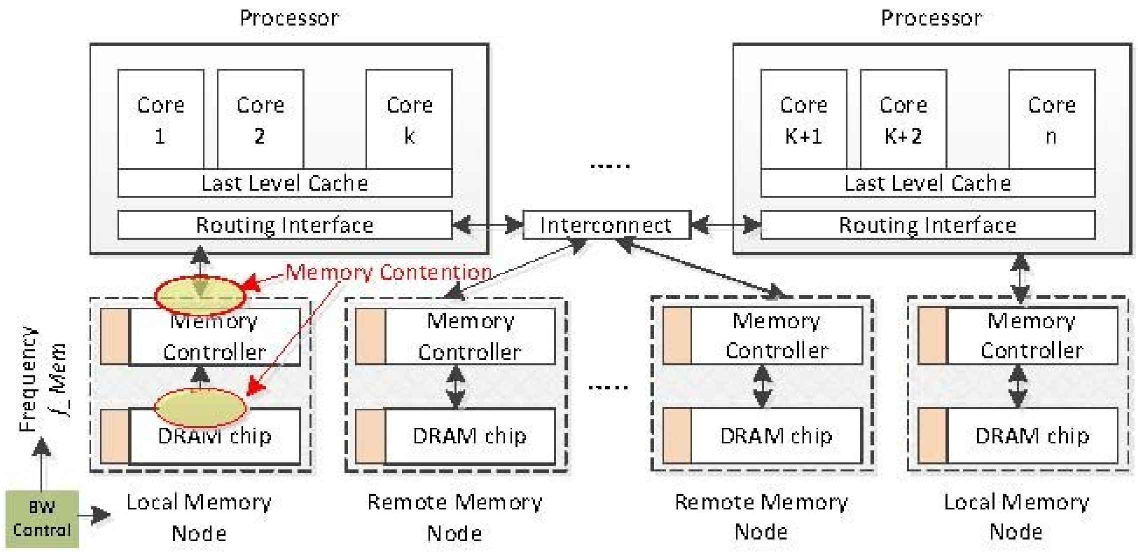


Figure 4.2-1 Memory systems in NUMA multi-core multiprocessors

In the system, each core in a processor accesses its local memory through the local MC. In addition, each core can also access remote memory through the routing interface and interconnect. Although the remote memory access can bring longer memory access latency than the local memory access, parallel memory accesses through multiple MCs increase total memory bandwidth.

The memory system performance can be impacted by both TLP and MLP. In particular, high level TLP could result in intensive memory requests, which in turn causes memory contention in multiple memory components (e.g., memory bus, memory controller, and DRAM chips), dynamic concurrency throttling (DCT) [52, 116, 178] is a technique to control TLP to control the tradeoff between performance and power.

DCT can be implemented by controlling number of threads and cores based on the need of application at runtime. MLP also has impact on performance of the memory system. By changing data distribution between memory nodes, MLP technique can control how many memory nodes should be involved in an application. MLP controls the available memory bandwidth to the application. Furthermore, memory frequency also impacts the memory system performance. Memory DVFS is a common technique to control memory frequency to alter memory system bandwidth. Previous work to improve memory efficiency has focused on dynamic change of TLP, MLP and memory frequency in isolation. However, these factors have interacting effects on memory performance and energy, which must be understood before devising an optimal strategy. In the next section, we introduce a customized M/M/C queuing model from queuing theory which naturally captures TL, MLP and memory frequency effects.

4.2.2. A M/M/C Queuing Model

Queuing theory is the mathematical study of waiting lines and queues. It has been widely used to address problems in traffic engineering and packet switching networks. To apply queuing theory, we abstract the memory system shown in **Figure 4.2-2**. It shows n cores try to access to the NUMA memory system through a simplified memory interface. The interface has a single logical queue connecting to m memory nodes. We use a M/M/C queuing system to model the above abstract memory system.

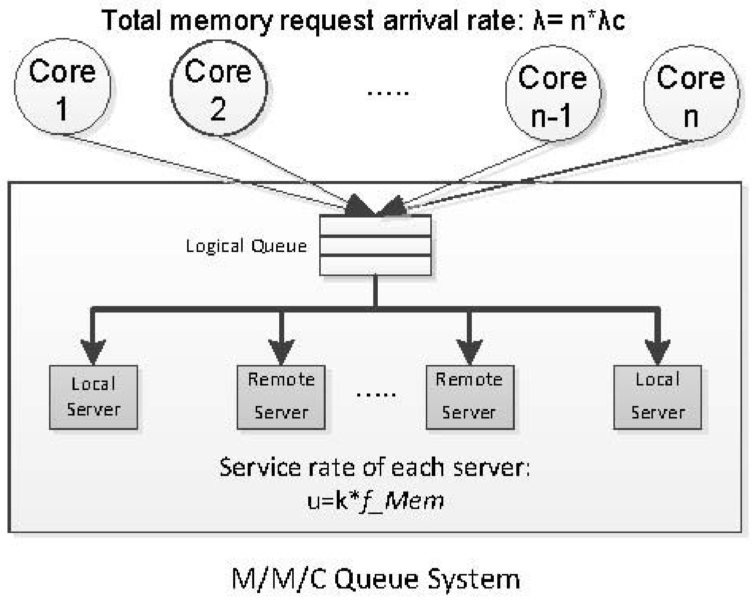


Figure 4.2-2 A simplified and abstract memory model to apply the M/M/C queuing model

The M/M/C queuing system models a single queue system with multiple servers to service multiple customers at the same time. The first two "M" in the M/M/C model indicates that all requests from customers follow the Markovian (the Poisson process), and the request service times have an exponential distribution. The "C" in M/M/C denotes the number of the servers in the queuing model.

In our cases, the server corresponds to the memory node; the customer corresponds to the processor core; and the customer request corresponds to the memory access request per core. We use the M/M/C system to estimate the average access latency of the abstract memory system shown in **Figure 4.2-2**. We also use the following notation to describe the system. The system has n active cores. The memory request arrival rate per core is λc. The system has m memory nodes running at frequency f_{Mem}. The maximal service rate of a single server is μ. μ is proportional to memory frequency, f_{Mem}. In other words, $u = k * f_{Mem}$, where k is a system-dependent constant factor.

We use the M/M/C system to model the average memory stall cycles per instruction as a function of the last level cache (LLC) misses. We denote the cycles per instruction as *CPI*. According to the classical M/M/C model, our M/M/C model includes both service cycles S_t and waiting cycles M_t :

Equation 4.2-1 : $CPI = S_t + M_t$

According to Kendall's notation [107] and the mathematical induction of the M/M/C model from [29], S_t and M_t can be expressed as follows:

Equation 4.2-2: $S_t = \frac{1}{\mu}$

Equation 4.2-3: $M_t = \frac{C(m,m\gamma)}{m\mu(1-\gamma)}$

In **Equation 4.2-3**, $C(m, m\gamma)$ is the Erlang formula[56]. It can be expressed as follows:

Equation 4.2-4: $C(m, m\gamma) = \frac{(m\gamma)^m}{m!(1-\gamma)} P_0(\gamma)$

Equation 4.2-5: $P_0(\gamma) = (\sum_{k=0}^{m-1} \frac{(m\gamma)^k}{k!} + \frac{(m\gamma)^m}{m!} * \frac{1}{(1-\gamma)})^{-1}$

γ in the **Equation 4.2-3** to **Equation 4.2-5** is an indicator in the M/M/C system to identify the system pressure. γ is defined as the ratio of the total customer (i.e., core) request arrive rate to the total service rate from all servers (i.e., memory nodes). When γ is close to zero, the number of requests from all cores is low, so the memory system has short access latency. When is close to 1, the system is saturated with memory requests which results in long access latency. Based on the definition of, it can be formalized as

Equation 4.2-6: $\gamma = \frac{n\lambda c}{k\mu}$

Given $u = k * f_{Mem}$, we further model γ as follow:

Equation 4.2-7: $\gamma = \frac{n\lambda c}{kmf_{Mem}}$

γ in the **Equation 4.2-7** contains four import factors that affect system performance. n represents the degree of TLP. Both n and λc capture intensity of memory requests. m represents the degree of MLP. Both m and f_{Mem} control available memory bandwidth.

Based on the **Equation 4.2-7**, we can formalize CPI as:

Equation 4.2-8: $CPI = \frac{1}{\mu} + \frac{C(m,mr)}{mu(1-r)} = \frac{1}{\mu}\left(1 + \frac{C(m,mr)}{m(1-r)}\right) = \frac{1}{\mu}\left(1 + \frac{(m\gamma)^m}{m!(1-\gamma)^2} P_0(\gamma)\right)$

Obviously, CPI is a function of γ. We now use CPI (γ) to represent CPI. To model the NUMA latency effect, we further introduce a term L_{numa} in the **Equation 4.2-1**.

Equation 4.2-9*:* $L_{numa} = \zeta \frac{R_{MA}}{All_{MA}} \Delta numa$

$$CPI = S_t + M_t + L_{numa}$$

$\frac{R_{MA}}{All_{MA}}$ is the ratio of remote memory access to the total memory accesses. $\Delta numa$ is the latency difference between local and remote memory accesses. ζ is a system parameter to quantify the impact of NUMA latency on the critical path. Based on **Equation 4.2-1** to **Equation 4.2-9**, we have CPI(γ) as follows:

Equation 4.2-10: $CPI\left(\gamma = \frac{n\lambda c}{kmf_{Mem}}\right) = \frac{1}{kf_{Mem}}\left(1 + \frac{(m\gamma)^m}{m!(1-\gamma)^2} P_0(\gamma)\right) + L_{numa}$

$CPI\left(\gamma = \frac{n\lambda c}{kmf_{Mem}}\right)$ in the **Equation 4.2-10** estimates the memory system performance by considering interacting effects from TLP, MLP and memory frequency.

4.3.Performance and Energy Models

The M/M/C memory model in the previous section models memory system performance. In this section, we propose a system-wide concurrency-frequency model (CFM) which incorporates the M/M/C memory model to enable performance prediction for an application. Then we use power models proposed by others [5, 23, 215] in conjunction with the CFM model to create an energy model to explore optimal energy consumption.

4.3.1. Concurrency-Frequency Model

The CFM model estimates the system-wide CPI of an application on multi-core NUMA systems by considering combined effects of TLP, MLP and memory frequency. The name Concurrency-Frequency Model captures both processor and memory-level concurrency (i.e., n and m) and memory frequency (f_{Mem}).

We first model the execution time (CPU cycles) of a parallel application using a single core (n = 1), and then extend it to model performance for multiple cores. The performance of a single core is modeled in the **Equation 4.3-1**:

Equation 4.3-1: $CYC_{Exec} = CYC_{CPU} + CYC_{MEM} = Inst_{CPU} * CPI_{CPU} + Inst_{Mem} * CPI\left(\gamma = \frac{n\lambda c}{kmf_{Mem}}\right)$

CYC_{CPU} is the total CPU execution cycles (i.e., with no stalls). CYC_{MEM} is the total CPU stall cycles due to accessing the memory system.

$Inst_{CPU}$ in the **Equation 4.3-1**is the total number of on-chip instructions, and $Inst_{Mem}$

denotes the number of instructions which cause the LLC misses. CPI_{CPU} is the average cycles of on-chip instruction. CPI(γ) represents the average memory stall cycles of a single core, which can be modeled in the **Equation 4.2-10** with $n = 1$.

We extend the **Equation 4.3-1** to model multi-core in the **Equation 4.3-2**. In general, $Inst_{CPU}$ can be divided into two parts: $Inst_{CPU\_P}$ represents the computation that can be executed in parallel without data dependency (e.g., independent floating point multiplication). Using multiple threads resident in multiple cores to execute this computation can potentially result in performance improvement. $Inst_{CPU\_NP}$ represents the rest of the on-chip instructions that has to be executed in serial (e.g., the critical section that has to be executed by individual threads).

Equation 4.3-2: $Inst_{CPU} = Inst_{CPU\_P} + Inst_{CPU\_NP}$

The parallel execution time can be described as:

Equation 4.3-3: $CYC_{Exec}(n, m, f_{Mem}) = \left(\frac{1}{n} Inst_{CPU_P} + Inst_{CPU_{NP}}\right) * CPI_{CPU} + Inst_{Mem} * CPI(\gamma = \frac{n\lambda c}{kmf_{Mem}})$

We divide the **Equation 4.3-3** by $Inst_{CPU}$ to calculate the system-wide CPI showed on the left hand side of the **Equation 4.3-4** (i.e., $CFM_{CPI}(n, m, f_{Mem})$).

Equation 4.3-4: $CFM_{CPI}(n, m, f_{Mem}) = CYC_{Exec}(n, m, f_{Mem}) = \left(\frac{1}{n} \frac{Inst_{CPU_P}}{Inst_{CPU}} + \frac{Inst_{CPU_{NP}}}{Inst_{CPU}}\right) * CPI_{CPU} + \frac{Inst_{Mem}}{Inst_{CPU}} * CPI\left(\gamma = \frac{n\lambda c}{kmf_{Mem}}\right)$

$CFM_{CPI}(n, m, f_{Mem})$ represents the CPI of an application when executing with n cores and m memory nodes at memory frequency f_{Mem}.

We substitute $\frac{Inst_{CPU_P}}{Inst_{CPU}}$ with α, and substitute $\frac{Inst_{CPU_{NP}}}{Inst_{CPU}}$ with (1 − α), so α denotes the ratio of parallel execution to the whole execution. We further $\frac{Inst_{CPU_{NP}}}{Inst_{CPU}}$ with β so β denotes the ratio of instructions that causes LLC misses to total instructions. Based on α and β, we rearrange the **Equation 4.3-4** as follows.

Equation 4.3-5: $CFM_{CPI}(n, m, f_{Mem}) = \left(\frac{1}{n}\alpha + (1-\alpha)\right) * CPI_{CPU} + \beta * CPI(\gamma = \frac{n\lambda c}{kmf_{Mem}})$

4.3.2. Energy Model

To calculate energy, we must know both execution time, $T(n, m, f_{Mem})$, and system power, $P(n, m, f_{Mem})$. The execution time can be directly derived from the **Equation 4.3-6**.

Equation 4.3-6: $T(n, m, f_{Mem}) = Total_{Inst} * CFM_{CPI}(n, m, f_{Mem})$

The system power includes CPU power (P_{CPU}), memory power (P_{Mem}) and other power (P_{Others}) consumed by other system components (e.g., disk and cooling fans). We assume $P_{Others}$is fixed, and focus on P_{CPU} and P_{Mem}, because CPU and memory account for major power consumption variance when changing *n, m,* and f_{Mem}. The system power is defined in the **Equation 4.3-7**.

Equation 4.3-7: $P(n, m, f_{Mem}) = P_{CPU} + P_{Mem} + P_{Others}$

To calculate P_{CPU}, we use a previously proposed model [2]. In particular, we first obtain the peak CPU power from vendor's CPU specification. Then we fix one half of the peak CPU power as static power, and scale the other half using *IPC*-based linear scaling.

To calculate P_{Mem}, we use a previous power model as well [5, 23, 215]. The memory power consists of three parts: 1) the background power that accounts for all static power when the memory devices stay in active and idle states; 2) the activate power that is the power consumed in the ACTIVATE command. The ACTIVATE command selects a row address from a memory bank and transfer the row's cell data to the sense amplifiers, putting the device into the active state; 3) the read/write power that accounts for the power consumption when data moves among the sense amplifiers, read/write latches and I/O pins. The memory power can be impacted by the memory frequency, number of memory nodes and total memory bandwidth utilization from the application (i.e., λ). We calculate the memory power using the Micron DRAM spreadsheet [2] and those parameters listed in **Table 4.4-1**. Note that the memory bandwidth utilization heavily depends on application memory access patterns. To accurately measure memory bandwidth utilization, we count the number of off-core memory requests using performance counters, and then divide this number by total CPU clock cycles. The energy is the product of time and power. We define the energy consumption of an application as follows:

Equation 4.3-8: $E_{App}\left(n, m, f_{Mem}\right) = T\left(n, m, f_{Mem}\right) * P(n, m, f_{Mem})$

Based on the above energy model, we define an energy improvement metric $EI_{App}(n, m, f_{Mem})$ as follows.

Equation 4.3-9: $EI_{App}(n, m, f_{Mem})$

$$= \frac{E_{App}(n_b, m_b, f_{Mem\,b})}{E_{App}(n, m, f_{Mem})} = \frac{CFM_{CPI}(n_b, m_b, f_{Mem\,b}) * P(n_b, m_b, f_{Mem\,b})}{CFM_{CPI}(n, m, f_{Mem}) * P(n, m, f_{Mem})}$$

$EI_{App}(n, m, fMem)$ estimates the ratio of energy of the configuration $(n, m, fMem)$ to the baseline configuration $(nb, mb, fMem\ b)$. The higher the value, the better of energy consumption improvement. We use it to search for the optimal energy configuration. Readers can use **Equation 4.3-6**and **Equation 4.3-7** to derive other energy-related metrics, such as the energy-delay-product (EDP).

Table 4.3-1 Parameter Description and Value

System	Description	Value
General Parameters		
n	Number of cores	
m	Number of memory domains	
f_{Mem}	Memory frequency	
Application-dependent Parameters		
CPI_{CPU}	On-chip cycles per instruction	
$CPI(r)$	Memory stall cycles per LLC	
$Inst\_CPU\_P$	instructions in parallel execution	
$Inst\_CPU\_NP$	instructions in sequential execution	
α	ratio of parallel execution of an application	
β	fraction of total LLC misses and on-chip inst.	
λ_C	bandwidth requests per core	
$\frac{R_{MA}}{All_{MA}}$	ratio of remote memory access	
Architecture-dependent Parameters		
PRE PDN	Mem. background power: precharge powerdown	14mW
PRE STBY	Mem. background power: precharge standby	168mW
ACT PDN	Mem. background power: active powerdown	28mW
ACT STBY	Me. background power: active standby	196mW
ACT	Activation Power	146mW
WR	Read and Write Power	448mW
k	Service factor of memory system	4
Δnuma	Diff. of NUMA Latency	45
ζ	System factor to estimate NUMA effect	0.4

4.4. Model Parameterization and Validation

In this section, we describe how to collect the values of the parameters listed in **Table 4.3-1** to use our models in practice. We first introduce the system setup and test benchmarks and explain how to use performance counter based approach to collect these parameters from a baseline run. Then we validate our models against direct performance and energy measurements.

4.4.1. System Setup and Test Benchmarks

We ran our experiments on an AMD X86-64 16-way system with 4 sockets, each of which has one quad-core AMD 8350HE Opteron processor. The CPU frequency is fixed to 2.0GHz. Each socket has a JEDEC-style, 16GB DRAM memory system with 2 DIMM channels. There is 64GB memory in total in the system. The memory system supports bus frequency scaling from 333MHz to 533MHz. We use PAPI version 5.1 to access the hardware performance counters to measure the performance events of applications. We study six representative benchmarks in the NPB benchmark suite listed in

Table 4.4-1 to validate our models. These benchmarks range from computation-intensive to memory-intensive, and exhibit diverse memory bandwidth utilization. We used the OpenMP implementation of these benchmarks. We compiled these benchmarks using Intel's Fortran compiler (version 12.0.2) with -O3 optimization. The operation system is Linux 2.6.1.

We vary three factors that affect memory performance: the number of cores (n), the number of memory nodes (m) and memory bus frequency (f_{Mem}). We control n and m through the OpenMP environment variables and linux numactl command. *numactl* runs applications with a specific NUMA scheduling or data placement policy. We use the interleaving policy to spread data evenly across memory nodes. The memory bus frequency can be chosen by setting BIOS at the system booting phase. We use two memory frequencies, 333MHz and 533 MHz, to verify the models.

Table 4.4-1 Collected parameters for applying the models

Program	Description	Class	α	β	λ_C	$\frac{R_{MA}}{All_{MA}}$	CPI_{CPU}
FT	Fast Fourier Transform	C	0.95	7.12E-04	4.20E-03	0.62	0.582
EP	Monte-Carlo methods	C	0.99	2.59E-05	2.56E-07	0.03	1.281
CG	Conjugate Gradient, irregular memory access	B	0.94	5.28E-03	7.65E-04	0.71	1.502
SP	Pena-diagonal matrices solver	B	0.91	2.31E-03	4.82E-03	0.72	0.953
MG	Multi-Grid on a sequence of meshes	B	0.88	2.01E-04	4.18E-03	0.64	0.984
BT	Block Tri-diagonal solver	B	0.98	2.01E-04	2.61E-03	0.67	0.85

4.4.2. Power Profiling

To measure system power, we use the WattsUp [20] power meter to sample the system power of each benchmark at runtime. We ran each benchmark with different number of cores (n) and memory nodes (m) configurations with the two memory frequencies. **Table 4.4-2** shows the power of three benchmarks at memory frequency 333MHz with different n and m configurations. We use the measured power and energy to validate our energy model.

Table 4.4-2 Power profiling with memory frequency set as 333MHz

Freq=333MHz	EP.C	Number of Memory Nodes				FT.B	Number of Memory Nodes				SP.B	Number of Memory Nodes			
		1	2	3	4		1	2	3	4		1	2	3	4
	16	569.1	567.1	567	568.1		586.8	603.4	622.7	647.7	16	582	593.4	621.9	633.4
	12	532.9	534.3	550.5	533.6		582.7	597.9	621.3	632.3	12	579.3	574.5	581.5	593.8
Num. of Cores	8	499.3	498.3	498.1	498.1		570.3	578.1	586.8	599.8	8	540.6	571	559.1	590
	4	461.6	460.8	460.2	459.8		540.9	565.3	568.9	567.3	4	512.1	522.1	521.6	532.8
	1	426.8	427.1	427.3	426.4		488.5	490.2	490.2	490	1	474.1	474.1	470.6	474.6

4.4.3. Determining Parameters

The CFM model in the **Equation 4.3-5** needs two sets of parameters. The first set contains architecture-dependent parameters. The first set includes $k, f_{Mem}, \Delta numa$. To calculate k, we execute the STREAM micro-benchmark [138] to stress the memory system to measure the maximal memory bandwidth (u) at two different memory frequency levels (333MHz and 533MHz). Given the memory frequency and $u = k * f_{Mem}$, we calculate k. To calculate $\Delta numa$, we also use STREAM. In particular, we first use numactl to control data distribution across memory nodes. Then we run STREAM and use the hardware counter event CPU READ COMMAND LATENCY NODE to measure latency differences between remote and local memory accesses. $\Delta numa$ is the average value based on the measurement. ζ is an empirical value based on our previous work [178]. ζ is an empirical scaling factor to calibrate the model's prediction and the measurement from systems. ζ is the is the average value from over 1500 exhaustive experiments with diverse execution patterns.

The second set contains application-dependent parameters. The second set of parameters includes $CPI_{CPU}, \alpha, \beta, \lambda_C$ and $\frac{R_{MA}}{All_{MA}}$. To calculate them, we use hardware performance counters. In particular, for each benchmark, we perform a baseline run with ($m = 1, n = 1, f_{Mem}$ = *333MHz*), and collect the following hardware counter events: total instructions, total CPU clock cycles, LLC misses and CPU stall cycles (PAPI_RES_STL). To calculate CPI_{CPU} (non-stall CPU cycles), we subtract CPU stall cycles from the total clock cycles. To calculate α, we measure total instructions spent in OpenMP parallel regions and sequential regions, and then calculate their ratio. To calculate miss ratios (β), we measure LLC misses and total instructions, and then calculate their ratio. To calculate the remote memory access ratio ($\frac{R_{MA}}{All_{MA}}$), we measure the requests of CPU to DRAM for a target memory node X.

4.4.4. Model Validation

We validate the execution time (i.e. **Equation 4.3-6**) and energy (i.e. **Equation 4.3-8**) against direct measurement at two memory frequencies (333MHz and 533MHz). This section shows some of the validation results due to space limitation, but the complete validation results can be found from our technical report. **Table 4.4-3** summarizes the absolute relative prediction error when changing the memory frequency.

Table 4.4-3 Relative prediction error with memory frequency set as 333MHz and 533MHz

Program	**Error**			
	333MHz		**533MHz**	
	Time	**Energy**	**Time**	**Energy**
FT	8.60%	14.20%	7.80%	11.40%
EP	3.50%	3.60%	4.80%	4.30%
CG	12.60%	14.90%	14.20%	13.40%
SP	14.60%	14.20%	12.80%	16.50%
MG	18.90%	19.40%	17.20%	17.40%
BT	10.70%	14.10%	13.50%	16.10%

We selectively show the benchmarks EP, FP and SP for further discussion of our validation because they consume small, moderate and large memory bandwidth, and hence are representative of diverse workload characterization.

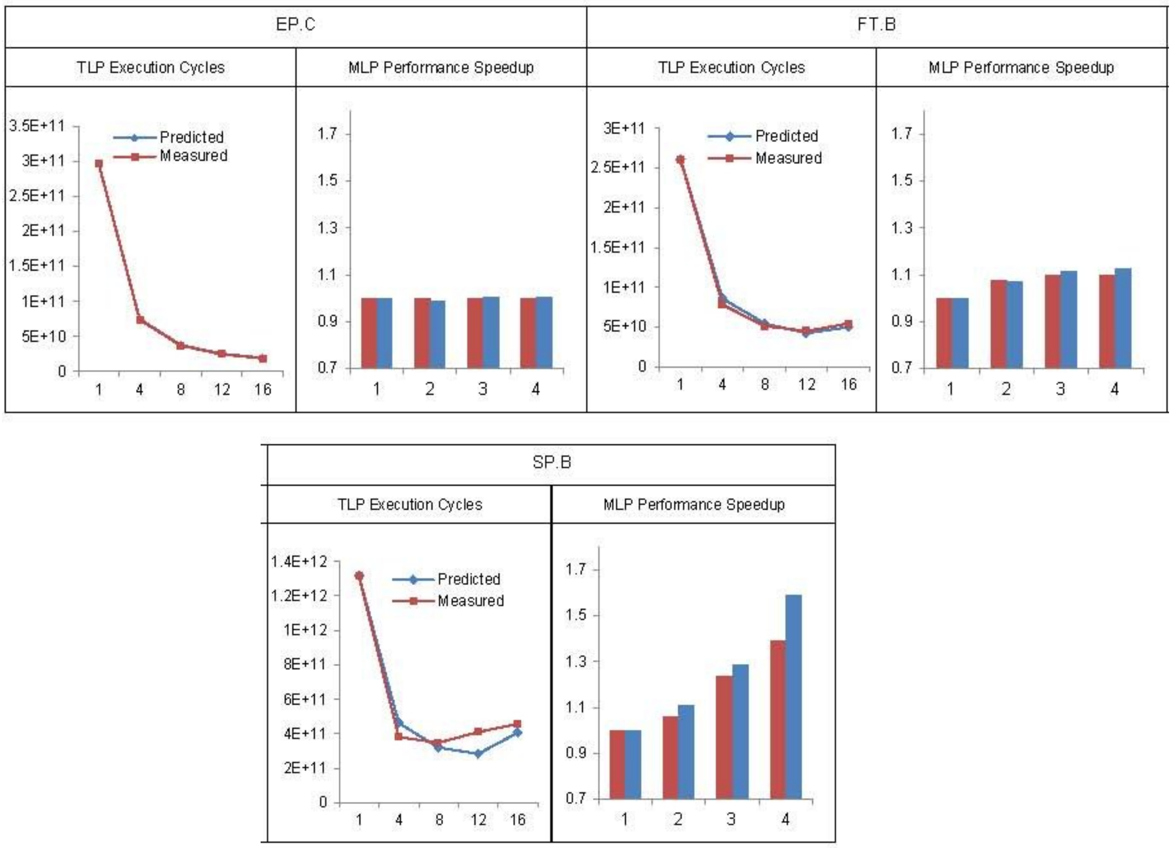


Figure 4.4-1 Validation of the performance model by varying TLP and MLP

The left figures of each benchmark in **Figure 4.4-1** shows the measured and predicted performance when we change the number of threads. These experiments use a single memory node to exclude NUMA effects. The figure shows that the prediction for EP achieves very high prediction accuracy. In addition, we notice EP has small β and $\lambda_C$values. The small β indicates that the benchmark has a small portion of memory instructions. The small $\lambda_C$indicates that the memory bandwidth utilization per core is low, so there are a small number of memory stalls. These two application-dependent parameters indicate that performance impacts from the memory system in the EP benchmark are small. For the FT benchmark, our model prediction is also very close to the measured value. We further notice that the best performance is achieved when the benchmark uses 12 cores. When the number of cores goes beyond 12, we found the execution time increases. This is an indication that the performance improvement due to the increase of TLP is outweighed by the extensive memory stalls due to the increase in memory access intensity. For the SP benchmark, we observe similarities to FT when we scale the number of processor cores. The best performance for SP is achieved at $n = 8$. We observe that the CFM model underestimates the impact of memory stalls for SP, but it still captures the general trends in performance.

The right figure for each benchmark on **Figure 4.4-1** shows measured and predicted performance as we change the number of memory nodes. These experiments use the maximal number of cores ($n = 16$) to stress out the memory system. We notice that we achieve high prediction accuracy for EP. Also, we find there is no performance improvement when increasing m. Because EP benchmark does not have intensive memory accesses, increasing m does not help to improve performance. For FT, our model also predicts its performance very well. According to our model predictions, FT can achieve 1.09x when the number of memory nodes is 4, close to the measurement (1.12x). For SP, the prediction accuracy is not as good as EP and FT. In particular, our model largely underestimates the performance improvement when the number of memory nodes is 4. This relatively low prediction accuracy may be due to the employment of the abstract memory system model in **Figure 4.2-2**. With the abstract memory system model required by the M/M/C model, there is only a single queue; in a real NUMA system, the memory system can have multiple physical queues, each of which corresponds to one memory controller. However, our prediction still accurately captures the performance trends as we vary the number of memory nodes.

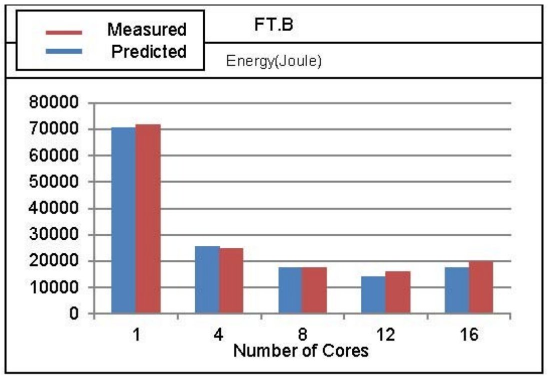


Figure 4.4-2 Validation of the energy model by varying TLP

Lastly, **Figure 4.4-2** shows the energy prediction accuracy for FT. We measure the energy consumption using 1,4,8,12,16 cores and 4 memory nodes at the memory frequency 333MHz. We compare the measurement with our prediction. The average absolute error rate is 11.4%.

4.5.Analysis

In this section, we use the verified results and model prediction to analyze the impact of TLP, MLP and memory frequency to performance and power by controlling n, m and f_{Mem}. Furthermore, we use the $EI_{App}$to explore the effects of the three factors to search for optimal energy consumption.

4.5.1. Impact of TLP

Figure 4.5-1 shows the impact of TLP. This figure shows the predicted CPI (for the application), and the measured and predicted memory stall (i.e., CPI(γ) for the memory system).

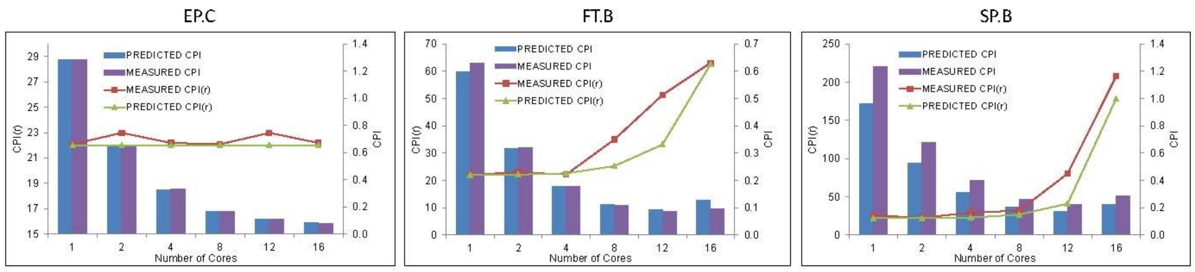


Figure 4.5-1 The impact of TLP to performance and average memory stall per LLC miss

We run the experiments using a single memory node at the memory frequency 333MHz. For CPI(γ) we use the PAPI RES STL performance counter to measure it. In the figure, we notice that the CPI(γ) with FP and SP increase as the number of cores increases. For FP, the predicted memory stall increases from 22 to 62. For SP, the predicted memory stall increases from 24 to 178 cycles. This is an indicator of memory contention, because the memory access becomes more intensive as the number of cores increases. For EP, however, we do not observe the increase in memory stalls. This is because the memory stalls depend on the memory bandwidth utilization per core (i.e., λ_c in our models). If λ_c is fairly small, (for EP, λ_c=2.56E-07), there is no serious memory contention along the data path, hence increasing TLP does not lead to an increase in memory stalls. We further notice that the performance of FT and SP degrades when the number of active cores increases from 12 to 16. This is an indication that the benefit due to increasing TLP is outweighed by the memory contention.

4.5.2. Impact of Memory Frequency and MLP

To understand the impact of frequency, we use the CFM model to predict the performance of FT at different memory frequencies and different levels of MLP. We normalize the performance to the baseline run ($n = 1, m = 1, f_{Mem} = 333MHz$). The results are shown in **Figure 4.5-2**

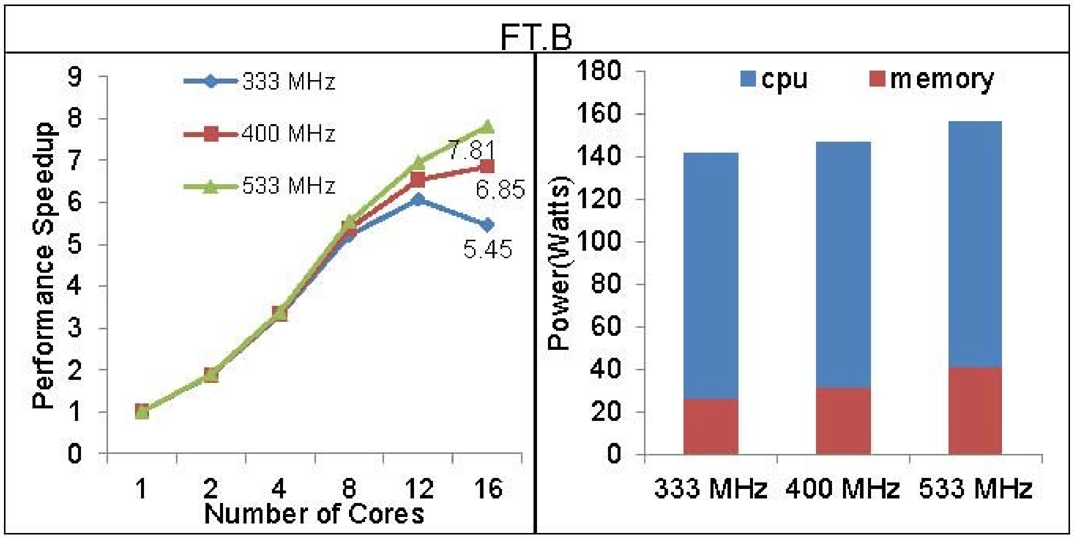


Figure 4.5-2 The impact of memory frequency to performance and system power

At 333MHz, the available memory bandwidth is insufficient to support memory requests from cores, so we have the worst performance at this frequency. The performance degrades due to increased memory contention when TLP is large (i.e. after $n = 12$). As we increase the memory frequency, the performance keeps increasing as the number of memory nodes increases. However, the increasing rate diminishes. This is an indication that the memory bandwidth improvement due to frequency scaling cannot keep up with the increase of memory access intensity as the number of memory nodes is increased.

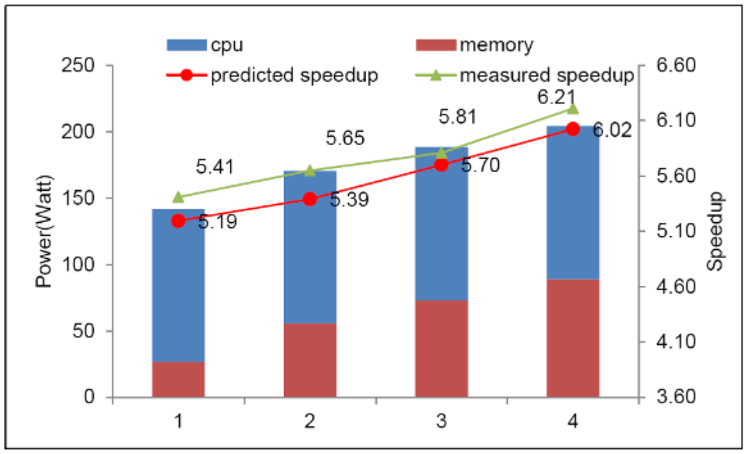


Figure 4.5-3 The impact of the number of memory nodes to performance and system power

Figure 4.5-3 shows performance and power consumption for the FT benchmark using 16 cores when increasing number of memory nodes from 1 to 4. We normalize the performance to one memory node. In general, as we increase the number of memory nodes, the performance increases from 5.45 to 6.13, and the power consumption increases from 141 Watts to 204 Watts.

4.5.3. Searching for Optimal Energy Consumption

We use the EI_{App} model to investigate energy improvement while varying MLP and memory frequency. We set the baseline configuration ($n_b = 1, m_b = 1, f_{Mem_b} = 333Mhz$) in the **Equation 4.3-9**. In the following analysis: we fix $n = 16$ with maximal TLP, and evaluate the optimal combination of m and f_{Mem} for the optimal energy consumption. **Figure 4.5-4** shows the EI_{App} for EP benchmark. EP is an embarrassingly parallel benchmark. It is highly computation-intensive with small memory bandwidth utilization. When the memory frequency scales from 333MHz to 533MHz, the EI_{App}goes down from 11.2 to 10.09. This means increasing memory frequency does not improve EI_{App} for EP. This is because computation dominates the most of EP execution.

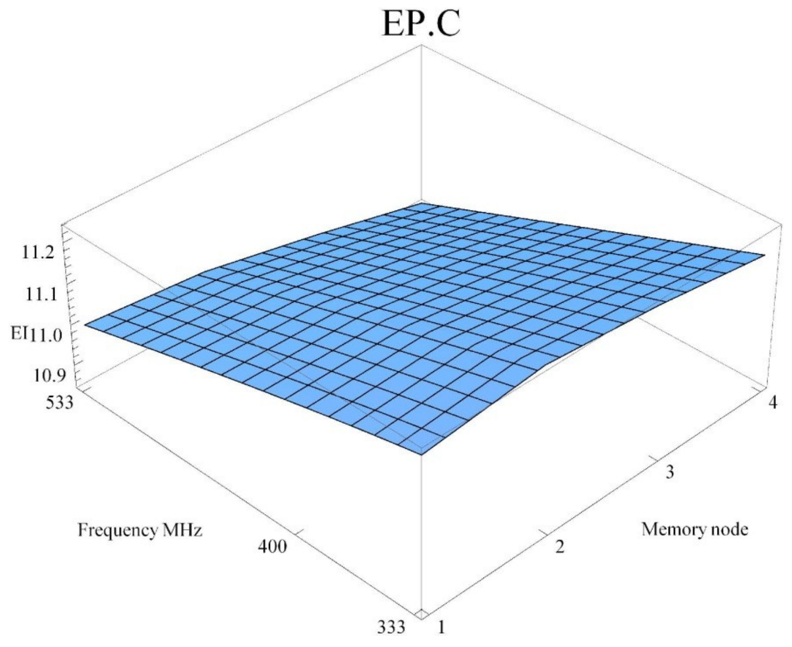


Figure 4.5-4 Energy improvement of EP benchmark

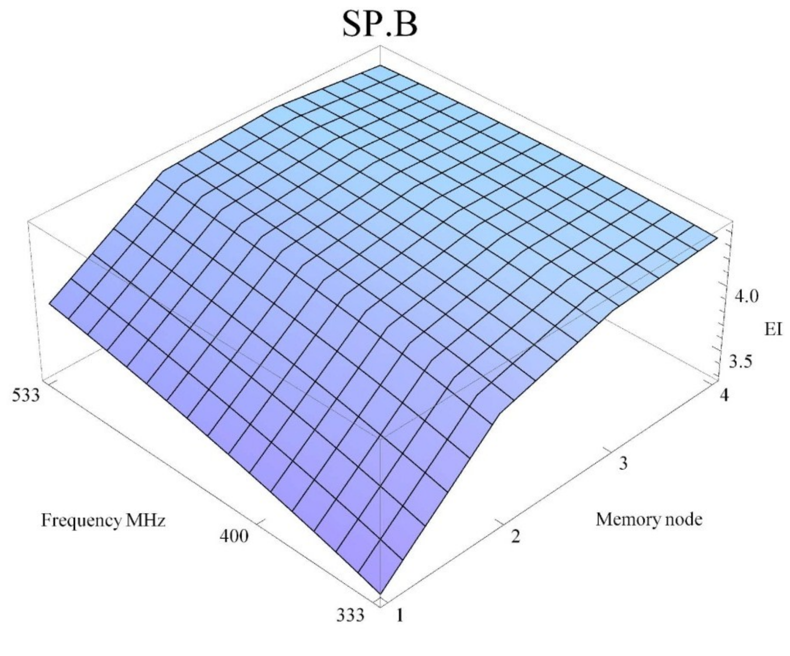


Figure 4.5-5 Energy improvement of SP benchmark

Increasing memory frequency does not improve performance but consumes more memory power due to rise of leakage power. In addition, increasing memory nodes from 1 to 4 reduces EI_{App} from 11.2 to 11.01, because the system consumes more static power when using more memory nodes. We also notice that scaling memory frequency has larger impact to EI_{App} than increasing the number of memory nodes. This is because increasing the number of memory nodes slightly increases background power consumption while leaving more memory nodes idle and it does not significantly increase memory power. Increasing memory frequency, however, can result in increase to both static power and dynamic power. The optimal EI_{App} of EP is at configuration (16, 1, 333MHz).

Figure 4.1-1 shows the EI_{App} for FT benchmark which has higher memory bandwidth utilization than EP. For FT, when the memory frequency scales up, the EI_{App} goes up from 4.54 (333MHz) to 6.36 (533 MHz) When we increase the number of memory nodes from 1 to 4, the EI_{App} improves from 4.54 to 6.05. The increased rate due to increasing number of memory nodes is less than that due to memory frequency scaling. We attribute this different increase rate to the NUMA effects to performance when accessing remote memory nodes. Increasing the number of memory nodes can potentially suffer from NUMA effects and degrade performance, hence negatively impacting energy improvement. In general, the EI_{App} model indicates the optimal $EI_{App}$of the FT benchmark is 6.36 using the configuration (16, 1, 533MHz). This configuration improves 40% EI_{App} compared to the configuration (16, 1, 333MHz). (i.e., 6.36 v.s. 4.54).

Figure 4.5-5 shows the EI_{App} for SP consumes more memory bandwidth than FT. The EI_{App} behavior of SP is different from EP and FT. When increasing memory frequency, the EI_{App} improves from 3.29 (333MHz) to 3.85 (533MHz). When scaling the number of memory nodes from 1 to 4, the EI_{App} increases from 3.29 to 4.35. We can see that the scaling of frequency and memory nodes result in similar improvement in EI_{App}. However, the optimal EI_{App} is 4.46 when the system uses 3 memory nodes at memory frequency 533MHz. This improves EI_{App} 35% compared to the configuration (16, 1, 333MHz), (i.e., 4.46 vs. 3.29).

Based on these analyses, we conclude that we cannot tune TLP, MLP and memory frequency individually to get optimal energy consumption. The optimal configuration depends on the application characteristics and the need to consider the combined effects of all three factors.

4.6.Conclusion

In this chapter, we first propose a concurrency-frequency model and an energy improvement model based on an M/M/C queuing model. The model predicts the application CPI as a function of TLP, MLP and memory frequency to estimate the system performance. Furthermore, our models show that the memory frequency, MLP and TLP have interacting effects on performance and energy. We validate our models against direct performance and energy measurements on an actual 16-way NUMA server. We demonstrate that the model-guided optimization can improve energy consumption up to 40% for applications that have high demand for memory bandwidth. The proposed model provides new insights that consider the interactive effects among TLP, MLP and memory frequency on performance and energy.

This page intentionally left blank.

Chapter 5

Heterogeneous Memory Controller Design

5.1.Introduction

The memory wall has long been a computing bottleneck, and it has been intensified by the introduction of multi-core processors. While the primary concern of the memory wall focuses on only bandwidth and latency, a new "power wall" challenge emerges for scaling out memory capacity within a reasonable power budget. When big data and HPC applications drive demand for memory capacity, traditional DRAM technology, unfortunately, with high static power will be less effective, and may not scale in terms of density and cost.

Previous work [66, 108, 144, 158, 159, 194, 196, 201, 209] has been proposed to address the power wall problem through heterogeneous memories by exploiting DRAM for performance and emerging NVRAM memory technologies, like phase change memory

(PCM), [199], STT-RAM[36] and memristors[49, 63], for capacity and energy efficiency. This work has proposed policies to control the trade-off between delivering performance and improving energy consumption on two basic memory organizations illustrated in **Figure 5.1-1**.

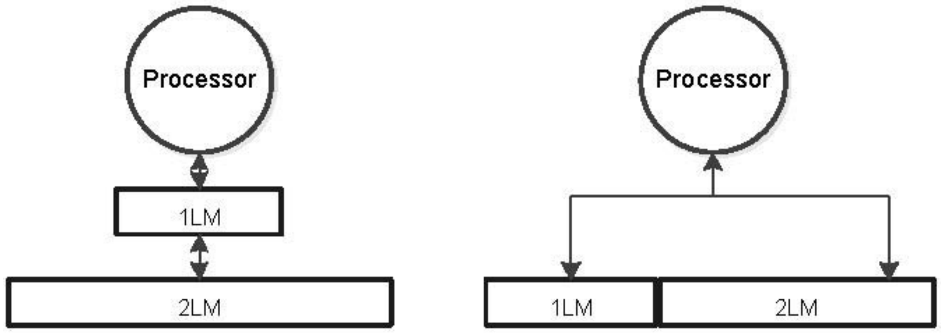


Figure 5.1-1 Candidate heterogeneous main memory organizations. (a) PCache: a hierarchical, inclusive system (b) HRank: a flat, exclusive system

Figure 5.1-1(a) shows a hierarchical, inclusive system. The first layer of memory (1LM) is used as a buffer for the second layer of memory (2LM). The 1LM space is usually invisible to the operating system (OS) and managed by the memory controller (MC)[158]. A few works [142, 158] introduce policies to manage data for this hierarchical design. These policies treat the 1LM as an associative cache and use LRU replacement to migrate pages. In this work, we call these types of policy "PCache".

Figure 5.1-1 (b) shows a flat, exclusive system. In this design, the 1LM and the 2LM have exclusive memory spaces. Both of the memory spaces are managed by the OS while the MC supervises the page migrations. Several works [159, 196, 209] have proposed policies to migrate pages in this flat design. These works use the following principles to design their migration policies: (1) place the performance-critical pages in the 1LM for performance and non-critical pages to the 2LM for low-power dissipation. (2) Rank pages based on the number of references and access recency. (3) Periodically migrate pages between the 1LM and the 2LM based on the ranking history. We call these types of policy "HRank". Although the above two types of memory policies show promising features for future computing systems, unfortunately, none of them are guaranteed to deliver and high performance and energy efficiency. As we shall demonstrate later, the effectiveness of these policies depends on the workload.

In this chapter, we propose HpMC, a new memory controller design which employs the hybrid use of the PCache and HRank policies to deliver better performance and energy based on system demand. The HpMC consists of a "Hybrid-policies Switching Engine" (HpSE), and several new components added to a vanilla MC to facilitate switching policies and migrating pages. In addition, HpMC implements an "Energy-aware Controller," (EaC). The EaC uses a locality engine, which periodically analyzes the degree of temporal locality based on reuse distance. If the degree of temporal locality crosses a certain threshold, it switches to PCache and switches back to HRank, or vice versa, to optimize energy consumption. We discuss the design of HpMC, including the switching mechanism in HpSE, the frame updating mechanism in the OS, and the potential cost of locality estimation in the EaC. HpMC is a hardware-software coordinating mechanism and manages pages without the limitations of previous work, including poor performance and energy caused by poor locality and high cost from updating mechanism.

We evaluate the HpMC on our trace-based simulator, HMSim. HMSim uses AMD SimNow[15] for the processor simulation to generate memory traces that feeds to DRAMSim[164] to get cycle-accurate memory performance and energy estimation. SimNow is a functional simulator, which enables 10-100x speedups over cycle-accurate approaches but lacks timing precision and accuracy. We propose a novel IPC calibration model to improve the precision and accuracy of the timing system in SimNow. In addition, we validate HMSim against two state-of-art native systems by comparing their bandwidth, latency and power.

Furthermore, we discuss our engineering efforts on re-architecting DRAMSim for the PCM system, and validate the PCM performance by comparing the numbers against others in the prevailing literature.

We use pF3D and LULESH, two representative HPC workloads, as our case studies to understand how the PCache and HRank policies impact performance and energy. The results demonstrate that both policies exhibit excellent performance and energy only for certain workloads. We further analyze the spatial and temporal localities of over 3000 diverse memory access patterns from the workloads of Coral Benchmarks[11] and lmbench [141], and use the analyzed results to build the switching rule for the EaC to optimize energy.

HpMC can be configured in three modes: HRank-only, PCache-only and EaC mode. We evaluate the HpMC performance using workloads from the Coral Benchmarks and lmbench running on a 4-way, out of order processor. We compare the bandwidth, energy and latency of HpMC using the three modes with a single layer DRAM system and a PCM system. The results show that the HpMC delivers better energy efficiency compared with its HM counterparts and improves energy consumption from 13% to 45% while providing almost the same bandwidth and larger capacity than the DRAM system.

This work makes the following contributions:

- We propose a new memory controller design which employs the hybrid use of the PCache and HRank policies to deliver better performance and energy. We conclude that no single HM policy delivers better bandwidth and energy. Our study demonstrates that better performance and energy can be achieved by hybrid use of these policies through a well-designed MC.
- We analyze the spatial and temporal localities of over 3000 diverse memory access patterns and identify the correlation between localities and energy consumption using two policies.
- We validate our simulation framework against two state-of-art native machines and

propose a novel timing calibration model to improve the accuracy of the simulation.

5.2.Hybrid Policies Memory Controller Design

Both PCache and HRank policies are designed to exploit the combination of two memory technologies into a single, heterogeneous system. They assume the 1LM is designed for performance, and the 2LM is designed for high-capacity and low static power. We assume DRAM technology for the 1LM and PCM technology for the 2LM in our work, but the flexibility of our simulation framework will support other emerging memory technologies, such as HMC [152], STT-RAM, [42], and memristors [49, 63].

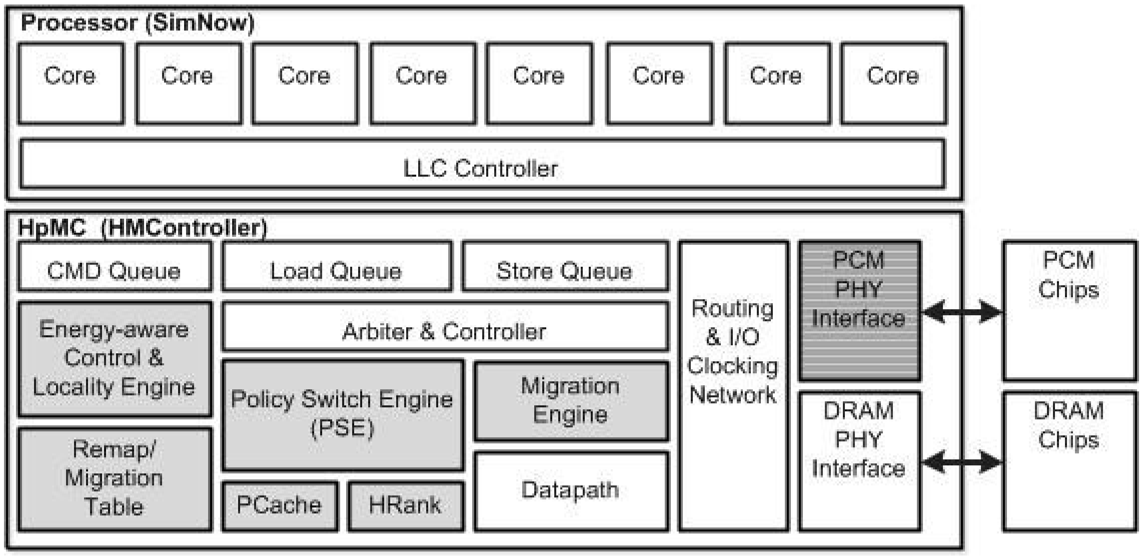


Figure 5.2-1 Hybrid Policies Memory Controller simulation framework.

A block diagram of the major components of our HM simulation framework, HMSim, is illustrated in **Figure 5.2-1**. In this section, we focus on the design of a hybrid policies memory controller, HpMC. The HpMC consists of two parts: HMController and DRAMSim [164]. HMController is an in-house, programmable, AMD SimNow [15] analyzer that SimNow loads into its execution environment. It is designed to process the read/write requests from the LLC controller and route requests to specific memory layers based on policies. HMController implements the two basic policies, PCache and HRank. It also implements a "Hybrid-policies

Switching Engine" (HpSE) and several logic blocks (illustrated in the shaded blocks in **Figure 5.2-1**) to switch policies, manage frames and update migrations to the OS. We start with a detailed implementation of the HRank and PCache designs. After that, we describe a few new components added to a vanilla MC to assist the HRank and PCache and facilitate the dynamic switching between them. Lastly, we discuss the DRAM and PCM physical interface, simulated by the DRAMSim that estimates the cycle-accurate memory performance and power.

5.2.1. Two Baseline HM Policies

1) PCache Policy: PCache is used to manage memory for the hierarchical, inclusive system. We base our design on the best available [158] which uses the DRAM as a hardware cache for the PCM. The PCM space is managed by the OS, and the DRAM is managed entirely by the MC without the OS involvement when a frame miss happens in the DRAM. The DRAM is implemented as an associative cache with an LRU replacement policy. On a miss in the DRAM, the frame that contains the cache line in the PCM will be brought to the DRAM. It uses an inclusion bit to indicate whether a frame holds a copy in PCM or not, and 8 dirty bits to track dirty sub-blocks of a frame. PCache adopts a lazy write-back strategy to reduce the write operations to PCM. In the lazy write-back strategy, when a frame is evicted in the DRAM, the write-back operation only happens when the inclusion bit is set to 0, or any of the dirty bits is set 1. PCache leverages a Remap/Migration Table in MC to keep track of the mapping between PCM frame IDs and DRAM frame IDs. When a memory request arrives, the MC checks the Remap/Migration Table to see if the requested frame is cached in DRAM or not. It also supports the line-level writes technique that the MC only writes dirty sub-blocks back to PCM to reduce the traffic [158].

2) HRank Policy: HRank is used to manage the flat, exclusive system. Our implementation leverages the idea of hot-cold frames [159, 196, 204]. Previous work uses a Multiple-Queue (MQ) ranking system [217] to rank the frames based on access recency and adjacency. In contrast, our HRank implementation ranks the frames according to the number of references. It updates the number of references of frames using a ranking list. For every 10 ms epoch, it re-selects the top-N hottest frames from the ranking list to move to the DRAM, and keeps the rest of frames in the PCM. It maintains a hot and a cold list to keep track of hot and cold frames. HRank compares the new ranking result with previous ranking result (in the hot, cold lists) and decides which frames to move in/out of DRAM and PCM. It then schedules migration frames to the queue of the Migration Engine. The HRank algorithm is simple but effective. It simplifies the design of MC since it only needs to update the references and rank/migrate frames every 10 ms. In contrast, MQ-based algorithms need to update the entire complicated MQ ranking system and decide page migration whenever a memory reference occurs[1]. HRank uses the Remap/Migration Table to keep the migration history. The MC periodically updates the migration history in the Remap/Migration Table of the OS to keep the system consistent and robust.

5.2.2. Hybrid Policies Switch Engine

1) Switching Mechanism: HRank and PCache have a fundamental difference in memory organization: PCache is an inclusive system in that the 1LM space is invisible to the OS, while HRank is an exclusive system that both 1LM and 2LM spaces can be seen by the OS. When HpSE switches from one policy to another, it needs to guarantee the OS is aware of the change to the inclusion/exclusion property.

[1] Ramos *et al.* improved the MQ algorithm by filtering out some rapid-fire accesses. However, it needs to control the filtering threshold to avoid hot frames stay in the PCM.

When HpSE decides to switch from PCache to HRank, 1)the HpSE interrupts the CPU and notifies the OS to update the page table entries (PTEs); 2) the OS replaces the old PCM frame IDs with new DRAM frame IDs in PTEs and flushes the corresponding TLB entries according to the Remap/Migration Table; 3) the HpSE frees the PCM frames stored in the Remap/Migration Table since the HRank does not need the PCM space to hold duplicates; 4) the HpSE cleans up the information in the Remap/Migration Table and uses it to track the migration history.

When the HpSE switches from HRank to PCache, it needs to change from the exclusive property to inclusive property. First, the HpSE notifies the Migration Engine to cancel scheduled migrations and cleans up the Remap/Migration Table. Second, the HpSE starts to the restore inclusive property. The HpSE checks the hot list in the HRank policy to get DRAM frame IDs and allocates unused frames in the PCM to restore the <DRAM,PCM> mapping in the Remap/Migration Table. If PCM does not have enough unused space to restore the inclusive property, the HpSE needs to vacate the least frequently used frames in PCM by checking the cold list for sufficient space. The HpSE then moves the vacated frames to a removing list. Lastly, the HpSE notifies the OS to replace old DRAM frames with new PCM frames in the PTEs based on the new Remap/Migration Table. If the HpSE sends the remove list information to the OS, the OS invalidates the corresponding PTEs in the page table and flushing TLB entries and programs the DMA engine to write dirty frames back to the hard disk.

5.2.3. Remapping/Migration Table

The Remapping/Migration table is used in both of the PCache and HRank policies. Each entry of the table has two columns to record frame IDs and several bits. In the PCache policy, two columns are used to track the mapping between DRAM frame IDs to PCM frame IDs. In the HRank policy, the two columns are used to track migration history. The first column records the source frame IDs and the second column records the destination frame IDs. The MC periodically updates the migration history to the OS to keep the system robust and efficient. Each entry has an inclusion bit and 8 dirty bits used in the PCache policy as described before.

In addition, we leverage other work [159] which uses two additional bits in the Remapping/Migration table for the communication between the MC and OS. The first is the Migrating bit. When the bit is set, it means that the frame is currently in migration status. The second is the Replacing bit, which is set by the OS when the OS is replacing the content of the frame. The OS is responsible for the Replacing bit and the MC is responsible for the Migrating bit. To maintain the robustness of the system, Replacing and Migrating bits are exclusive and cannot be set at the same time. The Remap/Migration Table is maintained by both the MC and OS. To guarantee the atomic operation on the table, the OS and MC use a memory-mapped register in the MC as the atomic operation token.

5.2.4. Migration Engine

The Migration engine uses a queue to record the scheduled migrations. It processes the migrations sequentially. In each migration, it reads the source frame into a buffer and sets the Migrating bit to 1. Once the Migrating bit is set, it writes the content of the buffer to the new destination and resets the Migrating bit when it finished. When a memory request arrives, it checks the Migrating bit to see if the frame is undergoing the transfer. During the migration, if the memory request is READ, it reads the data from the source frame; if the request is WRITE, the Migration Engine cancels the migration and finishes the write operation.

5.2.5. Energy-aware Controller and Locality Engine

The Energy-aware Controller (EaC) periodically uses the locality engine to keep track of the reuse distance distribution to calculate the degree of temporal locality, M_t. EaC switches between PCache and HRank policies to optimize energy based on the degree of the M_t. As we shall demonstrate later, the energy consumption of both policies has a strong correlation to the M_t. Reuse distance analysis is a popular tool for predicting locality and performance. However, several works [103, 169] have shown that the performance penalty is a major drawback of the tool in software-based, cache systems. Zhong and Chang report 2-4x slowdown using compiler-based instrumentation for single-thread benchmarks [41]. Schuff et. al. report averaging 29x slowdown with 19.6% sampling rate for multi-thread benchmarks [169].

We argue that the overhead of reuse distance in main memory is negligible and feasible for online estimation with the help of additional hardware for the following two reasons: 1) The traffic in main memory is 40-100x smaller than in the cache system. Thus, the estimation overhead can be drastically reduced. 2) Several stochastic models have been proposed to approximate the reuse distance online with small computation cost. Shen et. al. [171] proposed a small hardware analysis device with a stochastic model that maps cheaper time distance to a more expensive reuse distance within 1% prediction error. Their approach can achieve the reuse distance estimation within 3-8 us. Since the updating period of the EaC is 10 ms in our implementation, the computation overhead (3-8 us) for M_t is relatively small and negligible. For the space overhead, the locality engine uses sixteen 64-bit counters to estimate M_t. Each counter i stores the number of memory references that the reuse distance is between 2^i to 2^{i+1}. Thus, the space overhead is only 128Bytes.

5.2.6. DRAM and PCM PHY Interface

HMSim uses a trace-based approach to simulate the system. It has two steps. In the first step HMSim uses the HMController to collect memory traces of DRAM and PCM during the execution. In the second step, HMSim feeds the memory traces to the DRAMSim to analyze the performance and energy. HMSim leverages the DRAMSim to simulate the DRAM and PCM PHY interface.

DRAMSim is a cycle-accurate memory system simulator designed for modeling DRAM DDRx memory systems. It models a memory controller to issue commands to DRAM devices. The memory system contains load/store queues and a command queue, and maintains the bank states of all DRAM devices to simulate the performance. DRAMSim simulates different types of DDRx technology through device *ini* files, which parameterizes major characteristics of the DDRx timing mechanism. In addition, we re-architect memory array architectures in the DRAMSim to simulate the PCM memory system. For convenience in our discussion, we call the re-architected DRAMSim for the PCM system, PCMSim. The PCM performance simulated by PCMSim is discussed in the validation section.

5.2.7. Storage overhead

A key challenge for enabling high-performance heterogeneous memories is to design a cost-effective metadata system (e.g., Remapping/Migration table) at a fine granularity. **Table 5.2-1** shows the storage overhead of the HpMC with 1GB DRAM + 8GB PCM memory. For each component in the HpMC, HRank needs 9MB in total to maintain the hot, cold and ranking lists. PCache only needs the Remapping/Migration Table to track the DRAM and PCM mapping. Thus, we assume PCache use 0MB in **Table 5.2-1**. In addition, each entry in the Remapping/Migration table requires 55bits for storing tags (22*2 bits two column frame IDs, 1 inclusion bit, 8 dirty bits, 1 migrating bit, 1 replacing bit). The queue size of the

Migration Engine is 1.5MB (44bits for source and destination frame IDs * 256K queue size). Lastly, the locality engine needs 128 Bytes to track the reuse distance distribution in our implementation. The total storage overhead for 1GB DRAM + 8GB PCM memory setting is 12.26 MB. For fast access, we assume that the storage in all the components in the MC is made of SRAM.

Table 5.2-1 STORAGE OVERHEAD

Component	Storage overhead
HRank	9MB 16bits reference counter* (256K hot list + 2M cold list + 2.25M ranking list)
PCache	0MB
Migration Engine	1.5MB 44 bits * 256K entries
Remap/Migration Table	1.75MB 55 bit tags* 256K entries
Locality Engine	128Bytes 16 counters * 64bits
Total Size	12.26MB

5.3.Validation

5.3.1. Validation Challenge

HMSim uses AMD SimNow for the processor simulation. AMD SimNow is an x86-compatible, multi-core simulation platform. It is a functional simulator in that its device models maintain the program-visible machine state, but the device models abstract the timing feature for faster simulation of the entire computer system. Although HMSim can leverage faster simulation in SimNow to explore applications at large scale, accuracy is a major challenge.

The basic timing unit of SimNow is an instruction; all instructions are assumed to execute in the same amount of time and are one clock cycle in length. This assumption may overlook the performance when long latency events (e.g. memory and page fault) dominate the execution. It is common that intensive memory events may cause inevitable memory stalls, and memory latency is longer than one clock cycle. The assumption in SimNow that "IPC is equal to 1" for every application is over-simplified and can skew performance estimation.

Fortunately, SimNow provides an interface to set an IPC value for each application. We use the MEM.BW benchmark from lmbench to illustrate how the IPC setting affects memory bandwidth in **Figure 5.2-1**.

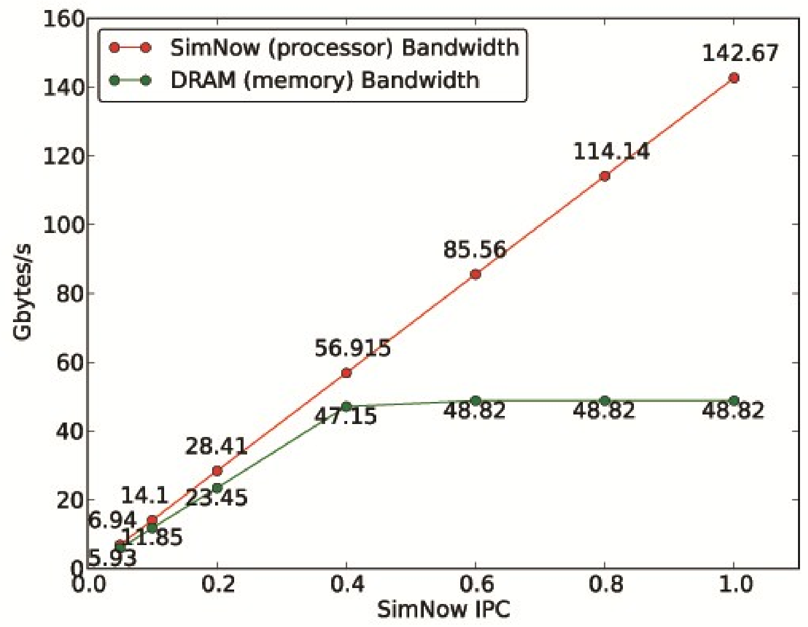


Table 5.3-1 The effect of the SimNow IPC on bandwidth

We ran the MEM.BW multiple times in HMSim simulating a single-layer DRAM system and manually varied the IPC from 0.05 to 1.0. In each run, we measured the SimNow LLC controller bandwidth and collected its memory trace. We fed the trace to DRAMSim to simulate the memory bandwidth of an 8GB, 4channel, DRAM DDR3-1333MHz system. The result shows that the bandwidth request from the LLC controller is proportional to the IPC settings in SimNow. In addition, DRAMSim result shows that the DRAM system bandwidth is saturated when IPC is greater than 0.4. This result indicates the IPC value configured in SimNow significantly impacts the demand for bandwidth from the LLC controller and the memory system performance estimation. Setting the IPC equal to 1 for all applications would lead to inaccuracies. For example, MEM.BW setting IPC to 1 generates 142GB/sec from the SimNow LLC Controller to the memory system (DRAM), and this number is far beyond what most contemporary computing systems can accomplish.

We also want to ensure the performance estimation of PCM system is within the ballpark of performance numbers found in the extant literature. Despite similarities to conventional DRAM memory array architectures, PCM requires solutions to several drawbacks before it can see widespread adoption as an alternative of DRAM, including long latency, high energy of write operations and limited endurance.

5.3.2. Methodology

We need to address both aforementioned challenges: 1) direct timing calibration against local systems, and 2) comparison to other prevailing systems mentioned in the literature. To address these challenges, we divide the validation into two parts. In the first part, we describe a timing calibration model to predict a proper IPC to calibrate SimNow performance. We then compare the simulated results after the model calibration with two native computing systems to show that the HMSim framework is sufficiently accurate for performance evaluation. In the second part, we discuss our efforts on re-architecting the DRAMSim for the PCM system, and validate the simulation performance by comparing with systems described in the prevailing literature.

The main architectural characteristics of the simulation framework are listed in **Table 5.3-2**. In this work, HMSim simulates a four-core, out-of-order processor equipped with an 8MB, 2-way instruction and data cache. The HMSim simulates a two-layer, heterogeneous memory system. It has four DRAM channels and four PCM channels; each channel has two DIMM ranks. In all simulations, it assumes no cold page faults and all data are all in 2LM. The DRAMSim and PCMSim use the close-page policy initially.

Table 5.3-2 SIMULATION SYSTEM CONFIGURATION

Feature		**Value/Configuration**	
Processor			
Processors (800MHz, x86-ISA)		4-way out-of-order processor	
I/D Cache		2-way, 128M lines, 64 Byte	
TLB		128-entries	
Cache block size/ page size		64 Byte/4KB	
Memory Systems			
Memory Controller		1333MHz, 4channels, 8KB row size, close page, Mapping Scheme 7	
Memory Devices (8x width, 1.5V)		DRAM	PCM[24], [35]
Delay	tRCD	15ns	55ns
	tRAS	36ns	71 ns
	tRC	51ns	126 ns
	tRP	15ns	55 ns
Current	Idd0	130mA	240mA
	Idd 2N	40mA	40mA
	Idd 3N	62mA	62mA
	Refresh	240mA	0mA

We validated HMSim performance against two native computing systems. The first system was a single-socket server with an 8-way (2x hyper-threading) Intel Westmere processor and an integrated memory controller supporting 3 memory channels of DDR3

DIMM; each DIMM has 2GB memory capacity. The second one is a server specifically designed for optimizing performance and energy efficiency. It has an 8-way (2x hyper-threading) Intel Haswell processor and dual-channel DDR3 memory system with 8GB capacity.

1) IPC Calibration Model:

We propose an IPC calibration model to calibrate the SimNow timing mechanism. The goal of the model is to predict a proper IPC value for each application to generate the same amount of demand for bandwidth from the SimNow LLC controller as that found in native machines. The model predicts the IPC of an application in SimNow by using the input from the native execution. The input includes the native measured IPC and a set of hardware event rates $(e1, e2, \ldots, en)$. We select events listed in **Table 5.3-4** that are critical to system performance, including the memory controller reads and writes, L1, L2, L3 hits, floating point, branch, and TLB misses. All selected events can be found in most contemporary processors. Each event rate, ei, is the number of occurrences of event i divided by the number of elapsed processor cycles during the execution. We model the SimNow IPC as a linear function of the native IPC and event rates:

Equation 5.3-1: $SimNow\ IPC = Native\ IPC * \alpha_0 + \sum_{i=1}^{n}(\alpha_i * e_i)$

We trained the relation in **Equation 5.3-1** with event coefficients $ai, i = 0, \ldots, n$ by using multivariate regression. We first collected the IPC, event rates and bandwidth from benchmarks listed in **Table 5.3-3** as training samples from native machines. We then ran the same benchmark on the HMSim using a single-layer DRAM and manually selected the SimNow IPC value that generates the same amount of bandwidth from the SimNow LLC Controller as the bandwidth collected in native machines to be the prediction target. We list the coefficients used in the model after training in **Table 5.3-4** . In the next section, we compare the performance of HMSim using the IPC calibration model with the two state-of-art computing systems described above.

Table 5.3-3 BENCHMARKS FOR EVALUATION

Program	Description	Source
CNS.STENCIL	A simple stencil-based test code for computing the hyperbolic components	ExaCT Co-Design Center
UMTmk	A microkernel performing three dimensional, nonlinear, radiation transport calculation CORAL Benchmark	CORAL Benchmark
Graph500	A scalable data generator and a BFS search kernels	CORAL Benchmark
MILCmk	A microkernel for the MIMD Lattice Computation (MILC) collaboration	CORAL Benchmark
AMGmk	A microkernel for parallel algebraic multi-grid solver for linear systems	CORAL Benchmark
LULESH	A proxy-app for the hydrodynamics simulation	CORAL Benchmark
pF3D	A parallel code for laser plasma interactions simulation	LLNL NIF
MEM.BW	A benchmark from lmbench to measure the memory bandwidth	lmbench
MEM.LATE	A benchmark from lmbench to measure the memory latency	lmbench

Table 5.3-4 IPC MODEL EVENTS AND COEFFICIENTS

Event	Coefficient	Event	Coefficient
IPC	0.36	FLOAT_INS	0.15
L1_HIT	1.31	TLB_MISS	0
L2_HIT	0.175	MEM_RD	5.07
L3_HIT	0	MEM_WR	10.7
BRANCH_INS	-0.66		

5.3.3. Performance Validation against Native Systems

We validated the performance of HMSim configured with a single-layer DRAM system using three benchmarks: MEM.BW, MEM.LATE and AMGmk by comparing simulated bandwidth, latency and power with the results measured in the two native systems. The memory bandwidth and latency validation with the Westmere machine and the memory power validation with the Haswell machine are discussed below:

***1) Bandwidth**:* We ran MEM.BW to compare the bandwidth in eight operation modes (rd, wr, cp, frd, fwr, fcp, bzero, bcopy, rdwr). We used the LIKWID [191]tool to measure the native memory bandwidth on the Westmere platform. The LIKWID tool counted the total number of DRAM CAS read and write commands issued on all channels from the integrated memory controller. Each DRAM CAS read and write command transfers 64 bytes of data (JDEDEC standard). The native bandwidth was calculated as the total transferred data size divided by the elapsed time. The top left chart in **Figure 5.3-1** shows the normalized bandwidth from HMSim and the Westmere system. The results are in different operation modes with an average error rate is 6.1%.

2) Latency: We used the MEM.LATE benchmark from lmbench and varied stride sizes from 64 to 4096 to validate the latency. On the Westmere system, we used the Intel VTune Amplifier [7] to measure the memory latency distribution using different stride sizes. According to the Intel spec[1], we excluded the events smaller than 32 cycles (i.e. oncore accesses) to ensure the memory accesses are all uncore events and calculated average latency. The top right chart in **Figure 5.3-1** shows the normalized simulated and native measured latencies. We normalized all simulated and native results to MEM.LATE.64 (i.e. stride size =64). In both simulation and native results, we can see the latency reduces when the stride size increases. This is because, when the MEM.LATE benchmark traverses same size of data, larger strides access the memory system less frequently than smaller strides and alleviate the waiting time in the transaction queue of the MC. Although the degree of degradation is different, we can see HMSim still gracefully captures the trend of degradation.

3) Power: The bottom chart in **Figure 5.3-1** shows the normalized power using MEM.BW, MEM.LATE and AMGmk. We ran AMGmk with three input sizes: 50,100, 200. We measured the native DRAM power through the Intel RAPL [59] on the Haswell machine. RAPL is a counter-based weighted model that estimates the DRAM power as a function of activity counters and pre-defined associated weights. The counters used in RAPL are described in a related paper [59]. DRAMSim uses a different method to model the power. It uses elapsed cycles and currents of different CAS commands to estimate power. Although they use different modeling approaches, the power estimation in DRAMSim still captures the trend as we measured in the Haswell. Our validation shows that, although the HMSim does not simulate identical results to those measured in native machines, it remains sufficiently accurate for performance evaluation.

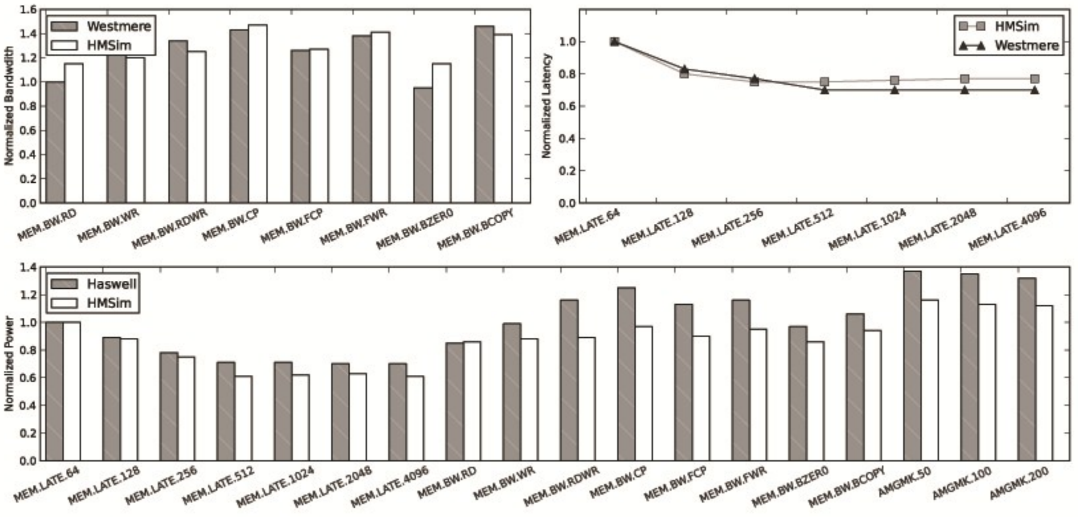


Figure 5.3-1 HMSim memory bandwidth, power and latency comparison with native systems

5.3.4. PCM System Validation

A PCM cell is a 1T/1R device, comprised of a storage resister and an NMOS access transistor. The storage resistor is typically a chalcogenide alloy. Ge2SB2Te5 is the most common material used in PCM. The PCM cell operates in two states. SET state represents bit value 0 and the crystalline phase while RESET state represents bit value 1 and the amorphous phase of the chalcogenide. PCM can be arranged in multiple level cells (MLCs) to store more states (phases) by applying different levels of heat that represent more bits. Our PCM system assumes three-bit MLCs, which provides four times the density of DRAM [27] [158] and gives the PCM similar cell area ($9F2\ -12F2$) compared to DRAM ($6F2-8F2$). Thus, PCM can leverage most CMOS peripheral circuitry used in traditional DRAM with minimal modifications. PCM cells might be hierarchically organized into ranks, banks, and blocks.

Despite similarities to conventional DRAM memory array architectures, PCM has several drawbacks including limited endurance, increased write energy and latency; these must be addressed before widespread adoption in hierarchical or flat memories. The techniques to address these drawbacks include wear-leveling techniques [156, 158, 216] to remove non-uniformity writes to prolong the lifetime of system; buffer reorganization techniques to improve locality and reduce the delay and energy gap [80, 111]; partial writes techniques [111] [89, 155, 204] to trace data modification to improve endurance and energy; and the PreSET technique [24], which executes a PreSET operation for a memory line as soon as the line becomes dirty in the cache. Thus, all PCM cells that are required in a write operation have been SET prior to the write to reduce the latency.

In this work, we built a PCM simulator, PCMSim. It re-architects DRAM memory logic from DRAMSim. PCMSim adopts buffer reorganization [111] and Data-Comparison Write (DCW) [204] techniques to improve PCM write latency and energy efficiency. We do not explicitly implement the wear-leveling since endurance is out the scope of this study, but buffer reorganization and DCW techniques may at least partially address the PCM endurance issue. In addition, we assume future PCM systems support a PreSET-like mechanism, in which case, the write latency is effectively the faster “RESET” latency instead of slower "SET" latency.

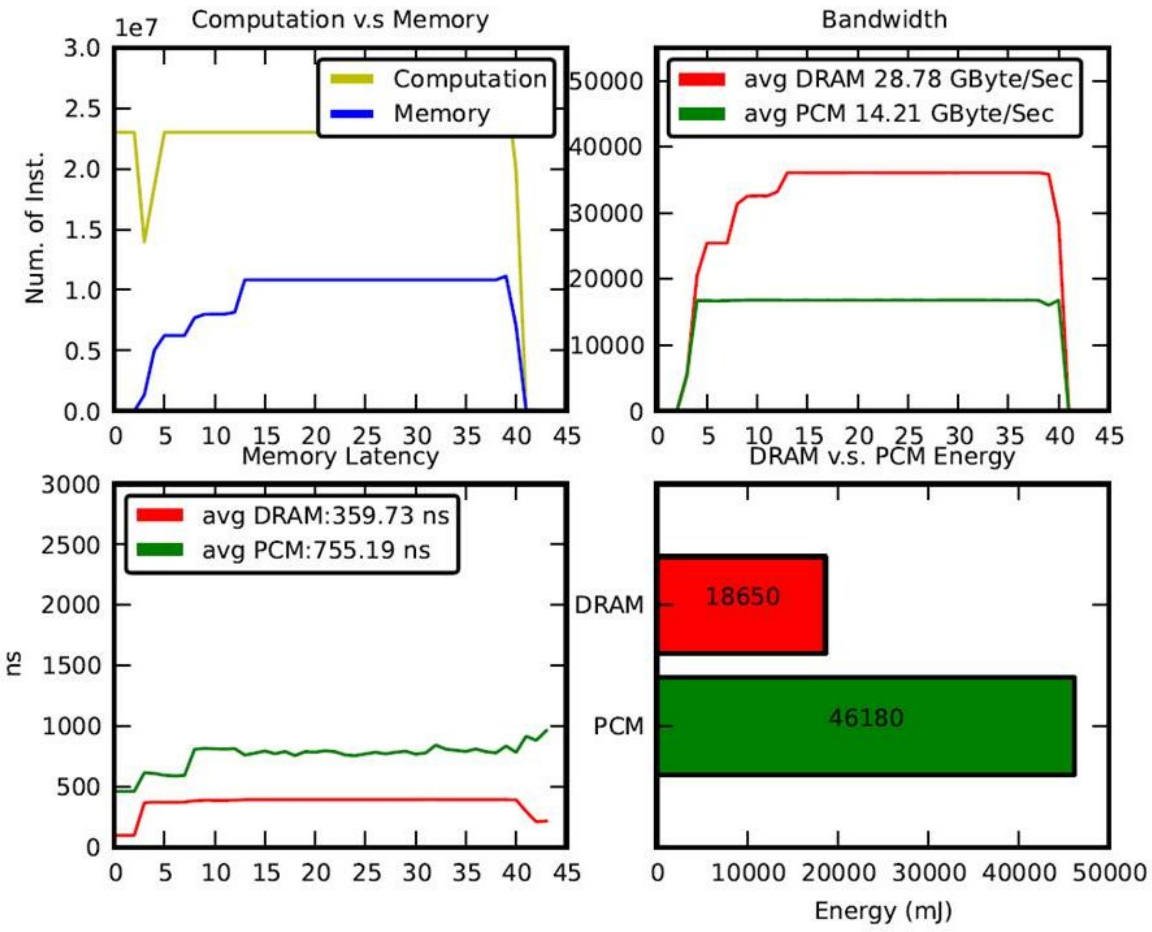


Figure 5.3-2 MEM.BW Performance on PCM and DRAM(DDR3) systems

We compare the PCM and DRAM performance by analyzing MEM.BW using the PCMSim and DRAMSim. Both memory systems used four channels and 8GB capacity. **Figure 5.3-2** shows the performance of the two systems. The x-axis of the first three charts represents elapsed epochs. The unit of the epoch is 10 ms. The top left chart shows the total number of computation instructions and memory instructions to interpret the ratio of computation and memory activities. The top right shows the bandwidth variation of DRAM and PCM systems over time. The average DRAM bandwidth is 28.8GB/sec, and the average

PCM bandwidth is 14.2GB/sec. The average DRAM latency is 359 ns, and the average PCM latency is 755 ns. The result shows that, under high-bandwidth conditions, DRAM outperforms than PCM. DRAM can support up to 2.0x bandwidth, 0.5x in access delay and consume 40% energy compared to PCM.

Table 5.3-5 PCM V.S DRAM PERFORMANCE CHARACTERISTICS

Performance	PCM	DRAM	Ratio	Range
READ latency	55 ns	15 ns	3.7	3 - 6
WRITE latency	55 ns	15 ns	6	5 - 30
READ energy	3.56 pJ/bit	1.04 pJ/bit	3.4	2 - 8
WRITE energy	12.35 pJ/bit	0.35 pJ/bit	35.5	10 -100

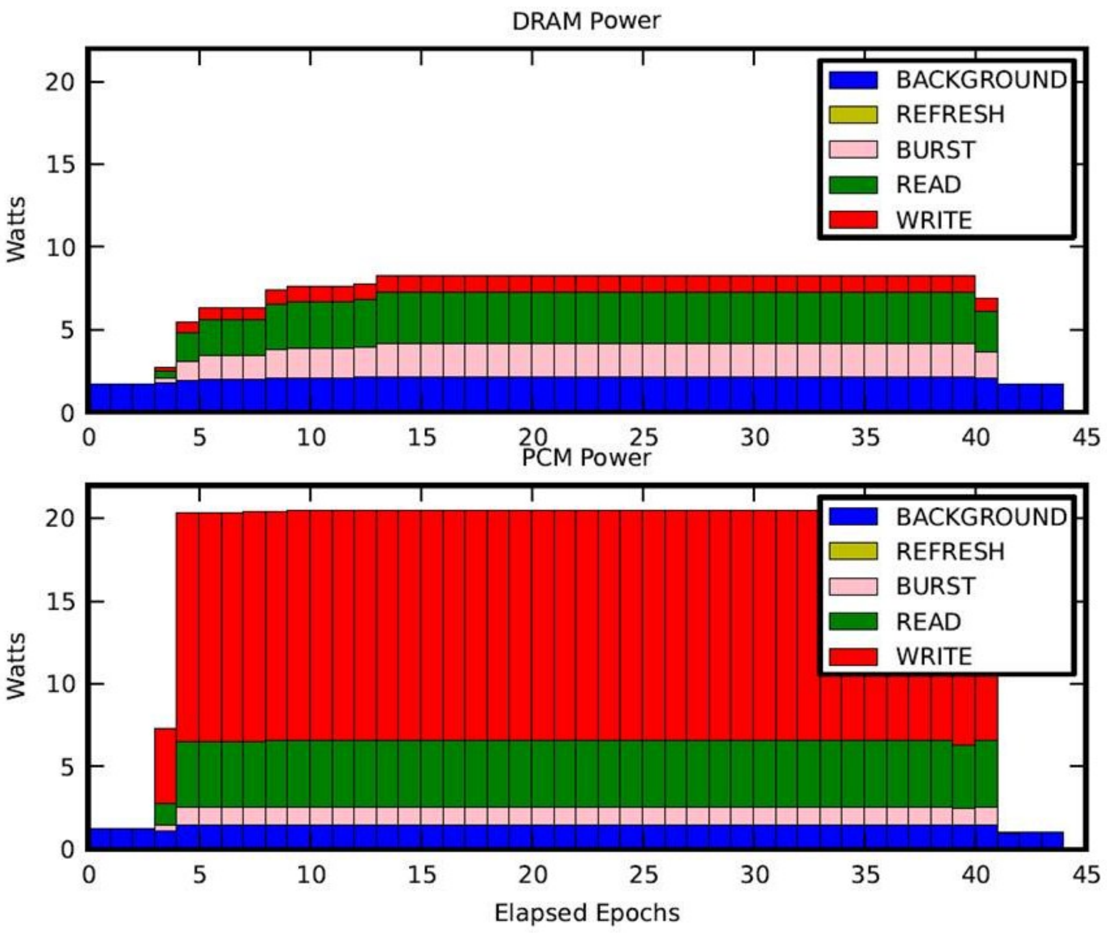


Figure 5.3-3 Power Breakdown of PCM and DRAM(DDR3) systems using MEM.BW

Figure 5.3-3 shows the power consumption of the two systems. The peak power of PCM is 20.1 Watts and 8.2 Watts for DRAM power. We further break down the power into background, refresh, burst, read and write power. The simulation results show that PCM dissipates less static power (i.e. background) than the DRAM system. We summarize the simulated performance of PCM and DRAM in **Table 5.3-5** and compare the numbers with recent literature [36, 85, 105, 111, 172]. The third column shows the ratio of PCM to DRAM performance in terms of latency and energy. The fourth column shows the range of the ratio suggested by the literature survey. **Table 5.3-5** shows the simulation results are within the ballpark of the recent literature survey.

5.4.PCache and HRank Policies Performance and Locality Analysis

We start with pF3D and LULESH, two representative HPC workloads, as case studies to analyze the bandwidth and energy of the HpMC using *PCache* and *HRank* polices. In addition, we analyze how dynamic spatial and temporal locality impacts energy consumption of both policies, and use the analyzed results to build the switching rule for the EaC to optimize energy.

5.4.1. Case Studies

pF3D: Figure 5.4-1 shows the pF3D performance of HpMC using *PCache* and *HRank* policies. From a bandwidth perspective, the average bandwidth of the DRAM and PCM in *PCache* is 8.17GB/sec and 0.76GB/sec. These values include the migration traffic between the DRAM and PCM. From the processor point of view, it can safely ignore migration traffic. Thus, we exclude the migration traffic to get the effective processor bandwidth. For the *PCache* policy, the effective bandwidth is calculated using the **Equation 5.4-1**. Based on the **Equation 5.4-2**, the effect processor bandwidth is 7.43 GB/sec in the pF3D (i.e. 8.17-0.76).

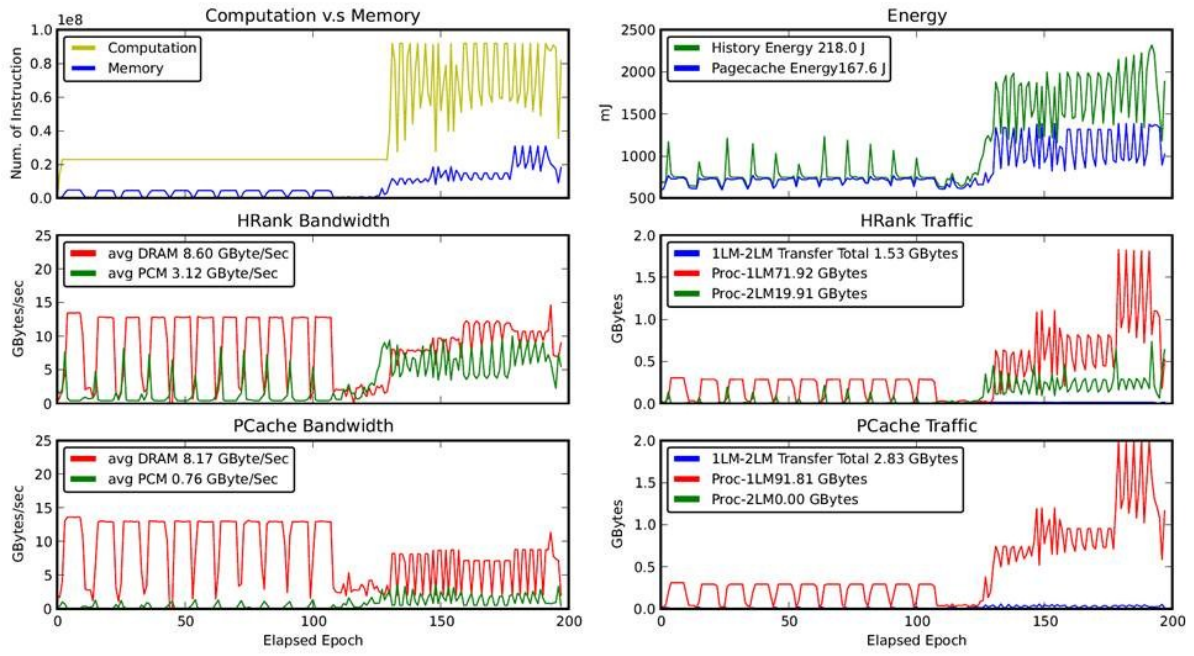


Figure 5.4-1 pF3D performance comparison using *HRank* and *PCache* policies in the HpMC

Equation 5.4-1: $BW_{PCache} = BW_{DRAM} - BW_{PCM}$

In contrast, the average bandwidth of the DRAM and PCM in HpMC using *HRank* policy is 8.6GB/sec and 3.12GB/sec. The effective processor bandwidth for the *HRank* policy is estimated by the **Equation 5.4-2**.

Equation 5.4-2: $BW_{HRank} = (BW_{DRAM} + BW_{PCM}) * \frac{\lambda_{DRAM} + \lambda_{PCM}}{\lambda_{DRAM} + \lambda_{PCM} + \lambda_{Migration}}$

λ_{DRAM} and λ_{PCM} represent the traffic between the processor and the two memory layers, and $\lambda_{Migration}$ represents the migration traffic between the DRAM and PCM. Thus, the effective processor bandwidth in *HRank* is 11.49 GB/sec (i.e. $(8.6 + 3.12) * \frac{71.92+19.91}{71.92+19.91+1/53}$). In the pF3D case, *HRank* provides more bandwidth than *PCache* since *HRank* allows the processor to access the PCM directly.

From an energy perspective, *PCache* consumes 167.6J energy during the execution and *HRank* consumes 218J. *PCache* use 24% less energy than *HRank*. When analyzing the traffic of the two policies, we found that, in *PCache*, the processor accessed 91.81GB of data in the DRAM while only 2.83 GB of data was from the PCM due to DRAM misses. In contrast, the processor accessed 21.44GB of data in the PCM and 73.45 GB of data in the DRAM when using the *HRank* policy. The additional memory accesses in PCM in the *HRank* system, lead to higher energy consumption for *HRank* versus the *PCache* in the pF3D case.

LULESH: Figure 5.4-2 shows the LULESH performance using both policies. Based on **Equation 5.4-3**.and **Equation 5.4-2**, *HRank* provides 7.99GB/sec effective processor bandwidth while *PCache* delivers 7.91GB/sec, similar to *HRank*. From an energy perspective, *HRank* consumes 20% less energy than *PCache* (157.3J vs. 192.6J). We further analyze the traffic between DRAM and PCM for the two systems. *PCache* needs to transfer 24.49 GB of data between the DRAM and PCM to meet demand; 38.17GB of data from the processor. The effective processor bandwidth is only 61% ($\frac{38.17}{38.17+24.49}$) of the total 1LM bandwidth. 39% of the 1LM bandwidth was used for data migrations between the DRAM and PCM. In the previous pF3D case, *PCache* only needs to transfer 2.83GB of data between the DRAM and PCM to meet 91.81GB processor demands. The effective bandwidth is 97% of the total 1LM bandwidth.

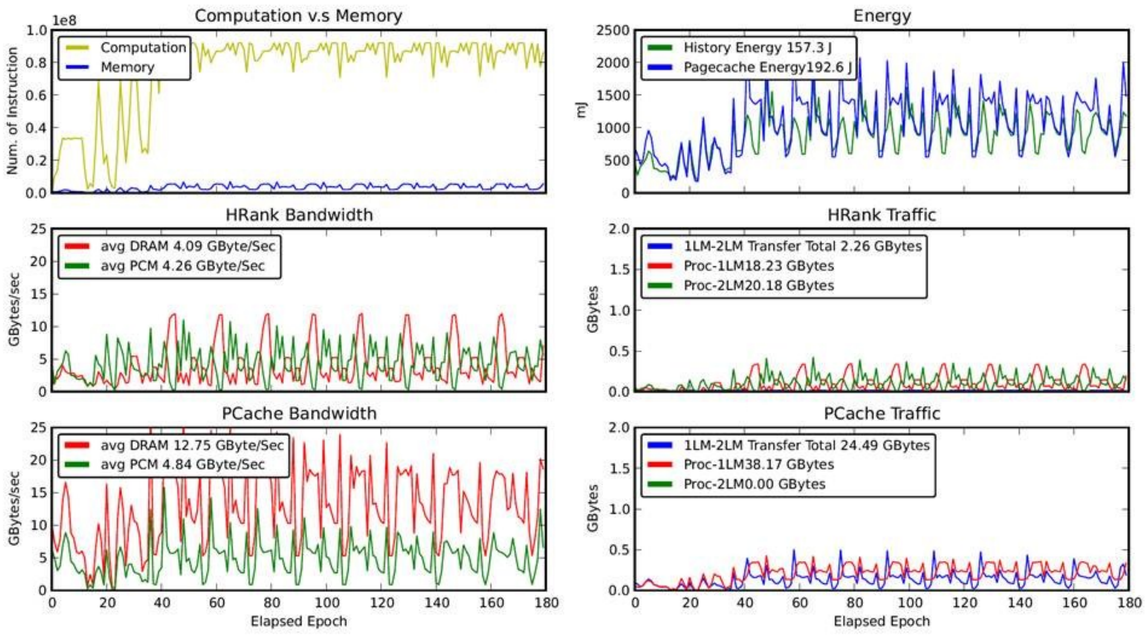


Figure 5.4-2 LULESH performance comparison using *HRank* and *PCache* policies in the HpMC

In conclusion, we found that for certain workloads, *HRank* provides extra bandwidth due to the direct access to the PCM. We also found that when the DRAM hit rate was low, the *PCache* policy using LRU replacement became too aggressive in migrating data between the DRAM and PCM. The aggressive LRU replacement dampened the effective processor bandwidth and wasted energy due to excessive migrations. In contrast, *HRank* effectively delays the migrations. The periodical migration strategy in *HRank* conserves more energy when the DRAM hit rate becomes low.

5.4.2. Locality Analysis for Energy Optimization

According to the above case studies, *PCache* leverages memory accesses with high hit rates in DRAM to conserve energy (pF3D). In contrast, *HRank* conserves more energy when DRAM hit rates are low (Lulesh). These findings lead to another consideration: can we leverage application locality to intelligently select the policy that conserves energy?

Hit rate can be characterized by two factors: access adjacency (spatial locality) and recency (temporal locality). To understand impact of locality on energy consumption, we use previously proposed metrics [197] to quantify the spatial and temporal locality of an application. The spatial locality is defined in **Equation 5.4-4**. $stride_i$ denotes the fraction of total memory accesses that are of page stride length i. An application that has all pages stride 1 references is assigned a value of 1; an application where half of the memory references are stride 1 and the other half stride 2 is assigned a value of .75, and so forth.

Equation 5.4-4 : $M_s = \sum_{i=1}^{\infty} \frac{Stride_i}{i}$

In addition, we quantify the temporal locality using the metric in **Equation 5.4-5**. The metric is based on the notion of the distance of data reuse. The reuse distance of some memory references to an address A is the number of memory references that have been accessed since the last access to A. In **Equation 5.4-5**, N denotes the longest reuse distance we traced (N= 512 in our study); $reuse_i$ is the temporal reuse function and represents the fractions of memory references with reuse distance less than or equal to i. The temporal locality metric, M_t, is less intuitive than the spatial locality metric. We can visualize that the M_s value estimates the area under the plot of the temporal reuse function, $reuse_i$, of the application. Since $reuse_i$ is monotonically increasing, the M_t value of an application that has more temporal locality is larger, because more memory references have low reuse distances.

Equation 5.4-5: $Mt = \sum_{0}^{log(N)-1} \frac{((reuse2i+1 - reuse2i) * log2N - i)}{\log_2 N}$

The M_t and M_s scores range between [0,1]. Higher scores mean better locality than lower scores. The access stride and reuse distance are inherent program properties and independent from any memory design. Thus, they are good indicators for program locality. To investigate the correlation of locality and energy consumption of two policies, we built a 2D (M_s, M_t) locality map for each benchmark listed in **Table 5.3-3**. Results for pF3D, LULESH, AMGmk, MILCmk, Graph500 and UMTmk are shown in **Figure 5.4-3**. The x-axis of each map represents M_s and y-axis represents M_t. A circle at position (x,y) on the map represents a small period (10 ms epoch) of execution in an application with M_s = x and M_t = y estimated from memory traces of the period. We also estimated the energy consumption of the epoch using *PCache* and *HRank* policies and chose the resulting lowest energy policy as the winning policy. If *PCache* wins in an epoch with the locality scores M_s = x and M_t = y, the map plots a green circle at position (x,y) on the map; if *HRank* wins, the map plots a blue circle. The size of the circle is used to represent the ratio of energy consumption of the losing policy to the winning policy. The bigger the circle, the better energy saving of the winning policy over the losing one. We analyzed diverse memory patterns from over 3000 epochs from the above benchmarks. In **Figure 5.4-3**, we see *PCache* wins for most epochs in pF3D, MILCmk and UMTmk. Since the circles in the MILCmk and UMTmk are small, there is not much energy difference between two policies. We also observed that blue circles dominate in the LULESH case while some green circles clustered on the top-right.

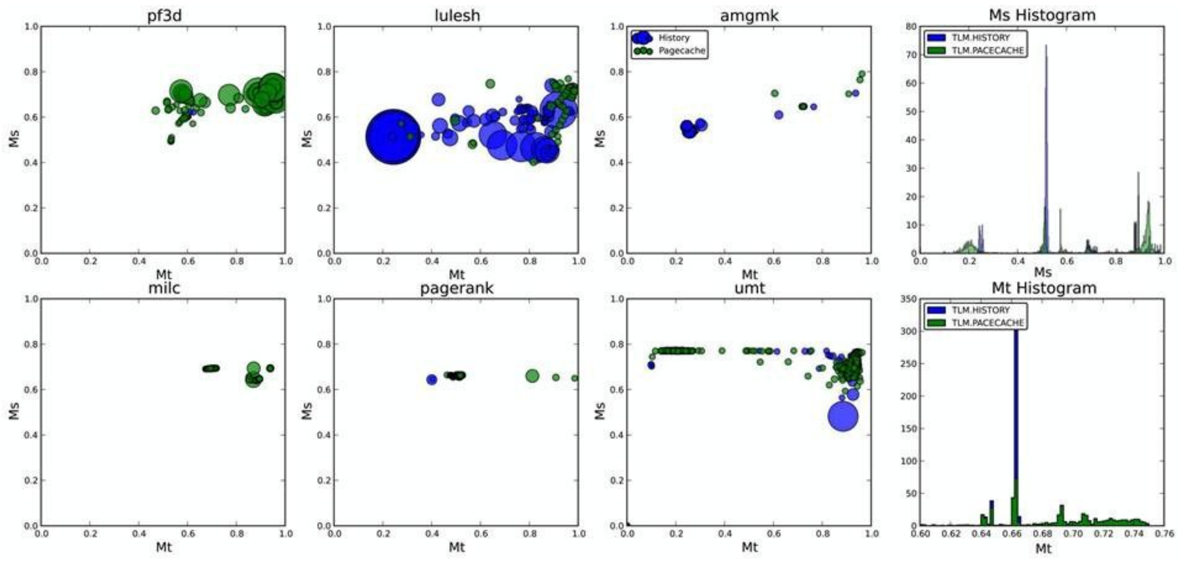


Figure 5.4-3 The correlation between locality and energy consumption of *HRank* and *PCache* policies

In addition, the rightmost charts show statistical histograms of occurrences of M_s and M_t values from the winning policy of all epochs in all applications. In the M_s histogram, we see both policies mixed spanning from 0.1 to 1. This result indicates that M_s is not a good indicator for selecting the winning policy. Spatial locality may be adversely affected by multi-core, out-of-order, parallel execution. In contrast, the M_t histogram shows that the *PCache* policy favors higher M_t (greater than 0.65) while the *HRank* system favors lower M_t (less than 0.65). When the M_t is high, the data in a page will be reused again soon and the page has a higher chance of remaining in DRAM without being evicted. In contrast, when M_t is low, the reuse of a page in DRAM is low, and the page has a higher chance of being evicted in the *PCache* policy, resulting in more energy consumed in migration. In this case, *HRank* can reduce energy consumption by less frequently migrating data and simply allowing the processor to access the PCM directly.

5.5.Results

5.5.1. Energy Optimization

We now show the energy consumption of EaC mode in the HpMC. EaC enables dynamically switching between *PCache* and *HRank* policies. The EaC periodically checks the temporal locality, M_t, and decides if it needs to switch to another policy or not. EaC sets the switching period to be 10ms and uses the locality engine to calculate M_t. Based on previous M_t histogram results, we build a switching rule as follows: If $M_t \geq 0.65$, EaC switches to *PCache* mode. If $M_t < 0.65$, EaC switches to *HRank* mode. **Figure 5.5-1** illustrates the energy consumption of *PCache* mode, *HRank* mode and EaC mode of pF3D and LULESH. The results indicate that the hybrid approach using EaC intelligently selects the low energy system over time with negligible prediction error. It improves energy consumption by 23%($\frac{169.3J}{218J}$) in pF3D and 20%($\frac{155.1J}{192.6J}$) in LULESH, compared with the worst case energy use.

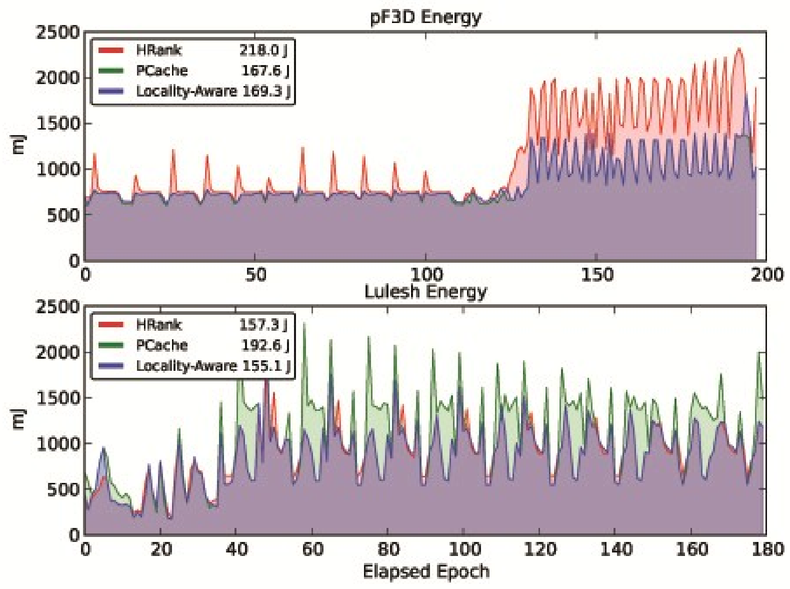


Figure 5.5-1 Energy consumption of *PCache*, *HRank* and EaC modes

5.5.2. Performance Evaluation

We evaluate the performance of the HpMC using three modes: *PCache*, *HRank* and EaC modes. We compare them with a DRAM-only system and a PCM-only system. The DRAM system uses 64GB, single-layer, 4-channels, DDR3- 1333 memory. The PCM system also uses a single-layer memory with 64GB capacity. We select 64GB for the DRAM and PCM systems as the base memory capacity since this is the common setting for state-of-art HPC systems. In the HpMC configuration, we use 8GB DRAM in the 1LM and 64GB PCM in the 2LM. The 1:8 ratio of 1LM to 2LM is our empirical selection which balances performance and energy consumption. We use the benchmarks listed in **Table 5.3-3** to evaluate the performance and the result is illustrated in **Figure 5.5-2**. We sorted the benchmarks based on DRAM system bandwidth from left to right.

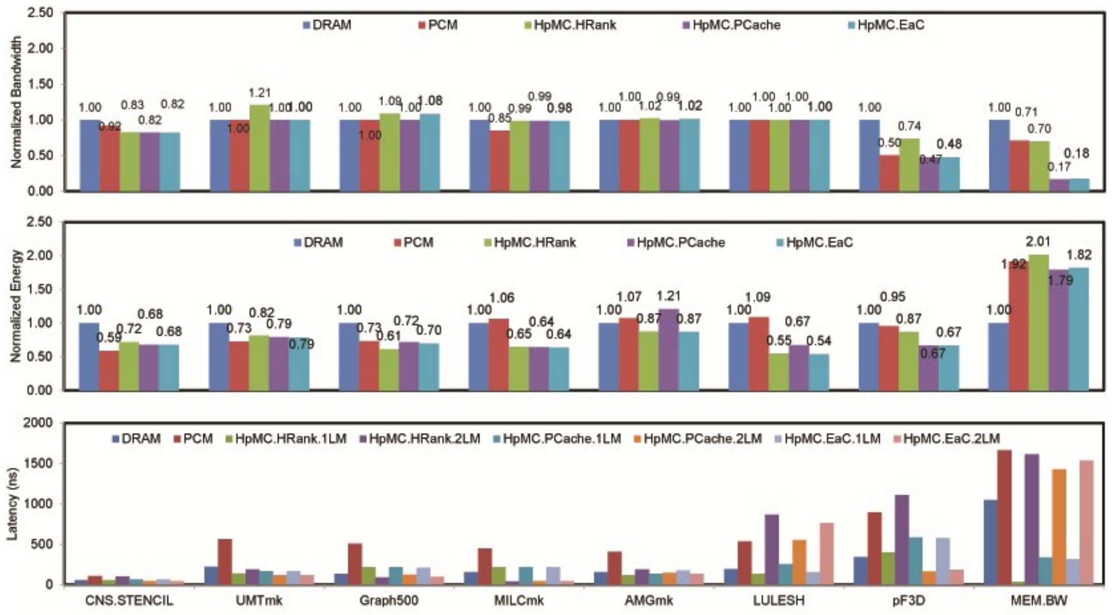


Figure 5.5-2 Bandwidth, energy, latency comparison of DRAM, PCM, and three modes in HpMC

The top chart in **Figure 5.5-2** shows the effective processor bandwidth calculated using **Equation 5.4-1** and **Equation 5.4-2**. We normalized the bandwidth to the DRAM system. The MEM.BW benchmark is used to measure the peak bandwidth of all settings. In MEM.BW, DRAM delivers best bandwidth performance, and PCM provides about 70% the bandwidth of DRAM. In HpMC, the *HRank* mode can deliver roughly the same bandwidth as the PCM while *PCache* has the worst bandwidth performance. This is because the MEM.BW benchmark is programmed to sequentially access memory addresses. Thus, the spatial and temporal localities are very low. In this case, *PCache* mode becomes inefficient since the policy design only allows the processor to access the DRAM, and much of DRAM bandwidth is used for data migration. Instead, *HRank* mode allows the processor to directly access the PCM and thus provides more bandwidth. We see the same phenomenon in pF3D, AMGmk, Graph500 and UMTmk. In UMTmk and Graph500, we found *HRank* bandwidth performance is even better than DRAM. This is due to more bank conflicts in the DRAM.

The middle chart in **Figure 5.5-2** provides the energy consumption. We normalized the energy to DRAM. In the low bandwidth scenario (i.e. applications on the left), PCM consumes less energy than DRAM due to less static power dissipation. When bandwidth increases from left to right, the energy ratio of PCM to DRAM also increases from 0.59 (CNS.STENCIL) to 1.92 (MEM.BW), because PCM uses more dynamic write power than DRAM. We found *PCache* and *HRank* modes conserved more energy than traditional DRAM in all cases except MEM.BW. The energy savings ranged from 13% to 45%. The *PCache* conserved more energy than *HRank* in MEM.BW, pF3D, MILCmk, UMTmk and CNS.STENCIL benchmark while *HRank* conserved more in LULESH, AMGmk and Graph500. In the EaC mode, we can see that EaC dynamically choses the lowest energy policies to optimize energy, however it may sacrifice performance on certain workloads (i.e. MEM.BW and pF3D).

The bottom chart in **Figure 5.5-2** reports latency of all memory systems. *.1LM and *.2LM represent the DRAM and PCM latency of the HpMC using three modes. We only model the latency of memory systems without consideration for the cost for MC migration and OS update since they are not always on the critical path of a memory access. In low and moderate bandwidth scenarios (i.e. CNS.STENCIL to AMGmk), we can see the latency of three HpMC modes are similar to the DRAM system. In the high bandwidth scenario (i.e. LULESH, PF3D and MEM.BW), we see the PCM latency in the three modes of HpMC is higher than the DRAM system. This is inevitable since the PCM is highly utilized; however, the latency can be hidden by a high degree of parallelism on the processor.

5.6. Discussion

5.6.1. Policy Details and Summary

PCache: The implementation of *PCache* policy is similar to the LRU policy in the systems. The migration unit of *PCache* is a page, not a cacheline. The cost of a page miss in the DRAM is high because the MC needs to migrate 4KB data per DRAM miss by default. *PCache* uses the lazy write-back strategy to trace the dirty sub-blocks and only write the dirty sub-blocks to the PCM. This strategy reduces the total migration traffic between the DRAM and the PCM. The lazy write-back strategy works well in cache systems; however, it could be a potential issue when we apply it to HM. The space overhead of the metadata system needed for the DRAM cache to trace dirty sub-blocks is significant. In addition, the MC needs fast access to the metadata to keep a low latency when the MC updates the state of pages in the metadata system. Recent works [96, 102, 128, 157] propose new designs using SRAM, 3D die-stacked DRAM to improve metadata system performance. This work only discusses the space overhead, but we need a deeper understanding of the metadata system performance. Some cost models are required to estimate the performance and the power of the metadata systems.

In addition, *PCache* relies on a high hit rate of the DRAM. When the DRAM hit rate is low, the DRAM will sacrifice a large portion of bandwidth for data migration, and this affects the available bandwidth for the processor. Based on our observation, *PCache* performs poorly for low spatial and temporal workloads because low locality means a low hit rate in the DRAM. We use pF3D and Graph500 to explain it. pF3D demonstrates high locality due to its stencil execution behavior. Graph500 exhibits low locality behavior due to its bread-first search kernels. **Figure 5.6-1** and **Figure 5.6-2** shows the traffic of Graph500 and pF3D. In Graph500, we see the HM system needs to migrate 10.97 GB size of data between the DRAM and the PCM (i.e. 1LM-2LM) for 55.57 GB of data from the processor to DRAM. The ratio of migration for Graph500 is 19.74 percent ($\frac{10.97}{55.57}$). On the other hand, we see the HM system only needs to migrate 2.83 GB size of data between the DRAM and the PCM (i.e. 1LM-2LM) for 91.87 GB of data from the processor to the DRAM. The ratio of migration for Graph500 is 3 percent ($\frac{2.83}{91.87}$).

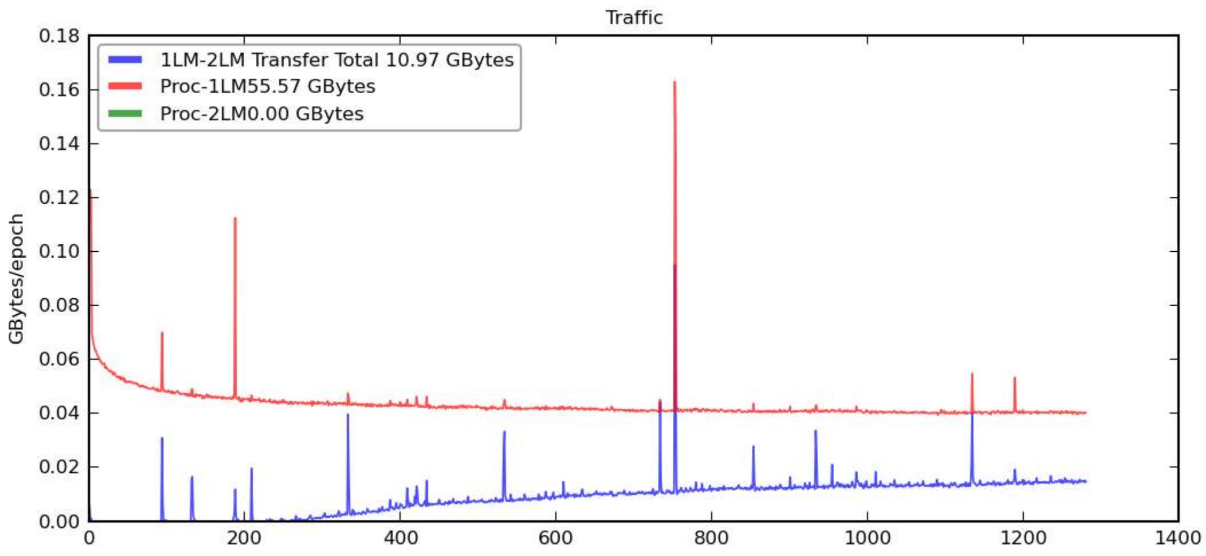


Figure 5.6-1 Traffic of Graph500 using *PCache* policy

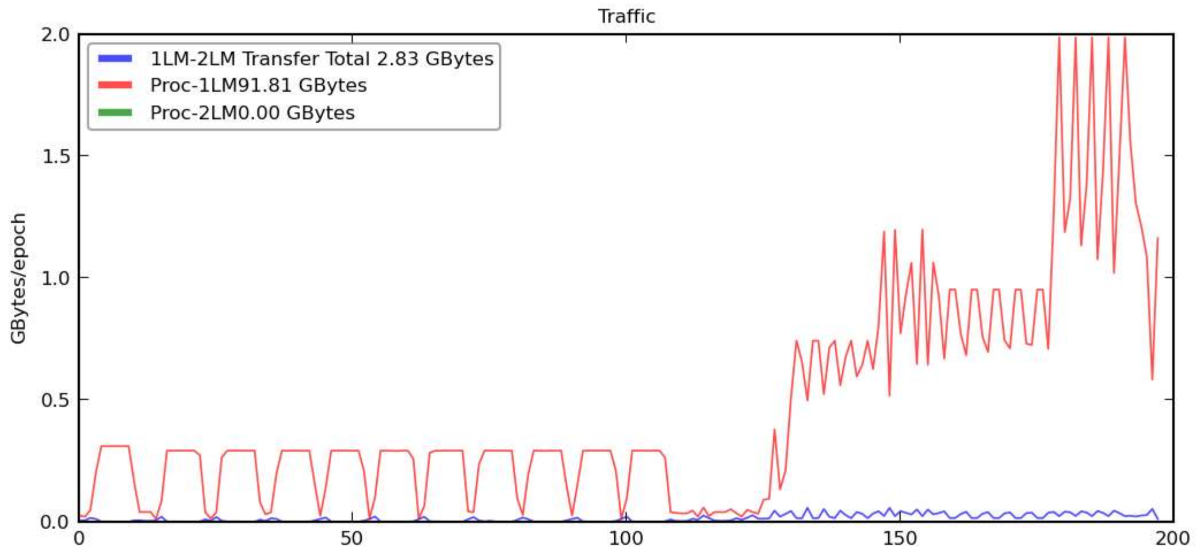


Figure 5.6-2 Traffic of pF3D using *PCache* policy

HRank: *HRank* policy uses a different strategy to migrate pages. It does not migrate a page when a miss happens in the DRAM. Instead, it periodically ranks the number of accesses of all pages and moves the performance-critical pages to the DRAM. The slow response strategy benefits low locality workloads because it lets the processor directly access the page in the PCM and avoids migrating the page to the DRAM, because the migration tends to be less useful due to low spatial or temporal locality. The epoch length is a major factor that affects the policy's performance. If the epoch length is too short, the *HRank* policy cannot trace enough ranking information to migrate performance-critical pages to the DRAM. In addition, the short epoch length also leads to performance overhead due to frequent migration. On the other hand, if the length is too long, it may cause the performance-critical pages to stay too long in the PCM, harming the overall system performance and consuming extra energy. The proper choice of epoch length is the key to performance and energy efficiency of the *HRank* policy. In our study, we manually selected 10ms as the default setting based on our system settings. However, the length of the epoch depends on several factors, including the DRAM size and the frequency of the memory bus. One of our future works will provide a sensitivity analysis for the epoch length to different system settings.

Summary: First, we found that, for high-temporal locality workloads, the MC needs to apply *PCache* policy to conserve energy; for low-temporal locality workloads, the MC can choose *HRank* to conserve energy. Second, the hierarchical, inclusive memory organization used for *PCache* policy only lets the processor access the DRAM. The PCM is used for migration when the DRAM misses or pages write-back. The processor cannot leverage the PCM bandwidth from this organization. On the other hand, the flat, exclusive organization used in the *HRank* policy provides extra PCM bandwidth for the processor due to direct access; however, this organization causes long access latency in the PCM. For HPC applications with high memory bandwidth requests, I suggest using flat organization with

HRank policy since the latency can be hidden by a high level of processor concurrency in the HPC systems.

5.6.2. The Impact of Memory Access Patterns

In previous sections, we demonstrated how we leverage the locality analysis to help HM systems improve performance and energy consumption. The variation of locality comes from different memory access patterns. In this section, we discuss how different memory access patterns affect the performance of policies. We use a matrix multiplication to illustrate the impact. Sparse matrix multiplication is the most time-consuming part in PDE (Partial Differential Equation) solvers, which are widely used in many HPC applications.

Matrix multiplication
Memory Access Pattern: Loops (i, j, k)

```
#define N 1024
# matrix multiplication (i, j, k)
for(i=0; i<N; i++)
for(j=0; j<N; j++)
for(k=0;k<N; k++)
{
    C[i][j] += A[i][k]*B[k][j];
};
```

Matrix multiplication
Memory Access Pattern: Loops (k, j, i)

```
#define N 1024
# matrix multiplication (k, j, i)
for(k=0; k<N; k++)
for(j=0; j<N; j++)
for(i=0;i<N; i++)
{
    C[i][j] += A[i][k]*B[k][j];
};
```

Figure 5.6-3 Pseudo codes of matrix multiplication using (i, j, k) and (k, j, i) memory access patterns.

Figure 5.6-3 shows two pseudo codes of matrix multiplication using two memory access patterns. The left codes use (i, j, k) order in the loops. The right codes use (k, j, i) in the loops. The two codes yield the same results in matrix C, but the right codes have a larger memory access stride than the left codes. It is easy for programmers to write their own codes similar to the example in the right if they do not consider the memory access patterns. However, it could lead to serious performance issues in HM systems.

The problem size of the matrix multiplication is 24MB. In our simulation, we set the DRAM size to 8MB and the PCM size to 1GB. This setting tries to make more misses in the DRAM. **Table 5.6-1** shows the performance of the matrix multiplication using two memory access patterns. In the (i, j, k) loops order case, we found *PCache* and *HRank* have almost identical performance. However, in the (k, j, i) loops order case, we observed that *PCache* has a significant amount of data migration traffic (i.e. 44.85 GB) between the DRAM and the PCM. This is because the memory accesses in (k, j, i) loops have large stride sizes and tend to cause more misses in the DRAM. One cacheline (i.e. 64 Bytes) miss needs to evict a page out from the DRAM and install a page from the PCM (i.e. 8KB). The miss penalty is very high. This example shows it is difficult for program developers to choose the best policy based on the memory access patterns in their code. It is better to have an interface, such as pragmas, APIs, or compiler speculations, to provide detailed memory access information for HM systems to decide migration policies.

Table 5.6-1 Performance of the matrix multiplication using two memory access patterns.

Memory Access Pattern	Policy	Bandwidth	Migration Traffic	DRAM Hit Rate
Loops (i, j, k)	*PCache*	9.7GB/s	0.02GB	0.99
	HRank	9.7GB/s	0.04GB	0.98
Loops (k, j, i)	*PCache*	9.6GB/s	44.85GB	0.81
	HRank	9.7GB/s	0.06GB	0.98

5.6.3. The Impact of Heterogeneous Memory to the HPC Systems

Heterogeneous memories provide an alternative choice for HPC systems to improve performance, capacity, and energy. It is still unclear how to manage pages in the HM system for the HPC community. This work provides insights on how we can leverage different management policies to benefit HPC workloads. For HPC application developers, they may want to have explicit control of page migrations to optimize the performance and energy consumption of their applications. However, poor page management leads to potential performance and energy loss. The matrix multiplication example in **Section 5.6.2** shows it is hard to rely wholly on programmers to manage pages to optimize performance and energy. Future HM systems should provide a set of migration policies for different execution requirements. In addition, the operating systems and programming models should also provide an interface (i.e. APIs or Pragma) for programmers. The interface can provide some details of memory access patterns in their applications and communicate with the HM system to choose the right policy.

5.7.Conclusions

In this work, we propose, HpMC, a new memory controller design which employs the hybrid use of the *PCache* and *HRank* policies to deliver better performance or energy based on the needs of a system. We also propose an energy-aware mechanism, which dynamically switches between *PCache* and *HRank* to conserve energy based on the degree of temporal locality. We compare HpMC with two single-layer, DRAM and PCM systems using the workloads from the lmbench, pf3D and Coral HPC benchmarks. The results show that the HpMC delivers higher energy efficiency compared with best available HM approaches and improves energy consumption from 13% to 45% while providing the same bandwidth and capacity of a traditional DRAM system.

We conclude that no single management policy delivers optimal bandwidth or energy. Our system demonstrates that better performance and energy can be achieved by hybrid use of these policies through a well-designed MC.

This page intentionally left blank.

Chapter 6

Conclusions and Future Work

6.1. Conclusion

The increasing number of available hardware resources on multicore, multi-memory architectures for high-performance systems have stimulated the research community to reconsider theories, techniques, and system designs to improve performance and energy consumption. More recently, many in the research community have discussed the introduction of heterogeneity to processors and memory systems to address the scaling challenges; however, such approaches introduce additional complexity into the system design. Resource management and energy-aware computation are two major challenges for scalable execution on emerging systems.

This dissertation presented a series of approaches and techniques to resolve the problems of resource management for energy efficiency on multicore, multi-memory systems.

6.1.1. NUMA Scheduling

We started with research on thread-management problems to improve performance and energy efficiency on NUMA systems. We presented a technique for determining the number of threads using an architecture-aware artificial neural network (ANN). We presented a critical-path, thread-mapping algorithm to minimize memory contention. We also presented a DyNUMA runtime system incorporating the above two techniques to dynamically manage the thread resources and validate the runtime for three diverse platforms.

We found that any attempt to throttle concurrency on a NUMA system after execution begins will redistribute computation between cores, thereby forcing extraneous cache misses, remote memory accesses, and contention. Prior work on DCT overlooked this problem, resulting in serious performance and energy loss. We extended the previous IPC-based DCT performance model and propose a new architecture-aware, ANN model to predict the thread concurrency. The model mapped the topology of the ANN to the NUMA architecture to capture the performance variation due to different thread mappings. In addition, we invented a heuristic thread-mapping algorithm to determine the best mapping configuration to minimize the memory contention and optimize performance and energy. We implemented a DyNUMA runtime system that employed the ANN predictor and thread-mapping arbiter in conjunction. The runtime system automatically changed the thread concurrency and mapping during the execution. We evaluated the runtime system using the NAS and Sequoia Benchmarks on three different NUMA platforms. Our runtime achieved an 8.7% improvement in wall-clock time on average, 16% improvement in EDP, and 12.3% improvement in MFLOPS/Watt.

6.1.2. NUMA Modeling

Next, we presented a novel analytical model for NUMA memory systems using queuing methods. Previous work based on the ANN and critical-path thread mapping techniques only considered performance and the energy impact of thread resources. The new models also took memory resources into consideration, including memory-level parallelism (MLP) and memory frequency. The models considered the combined interactive effects of these factors on system performance and energy, and overcame the limitations of previous works wherein they only considered these impacts in isolation. The model can help system resource managers to understand the tradeoff between performance and energy in a broader perspective. We investigated and evaluated the model on a 16-way multicore NUMA platform. We showed that significant energy benefits can be brought about from concurrency throttling, MLP throttling, and DFS.

We first investigated the memory system design on modern multicore NUMA systems. We found that the memory system performance is determined by three important factors, namely thread-level parallelism (TLP), MLP, and memory frequency. In particular, high-level TLP results in intensive memory bandwidth, which in turn causes memory contention in multiple memory components, such as the memory bus, memory controller, and DRAM chips. In addition, MLP also affects the performance of the memory system. MLP determines the theoretical bandwidth's upper bound; by changing the data distribution between memory nodes, MLP techniques can control how many memory nodes should be used during the execution. Last, the memory controller frequency decides how soon a memory request can be served, and this affects the memory latency and bandwidth. Unlike previous models that have analyzed single factors in isolation, our queuing models predict the combined effects of the three factors mentioned above, which dominate memory performance. We validated our model against a 16-way system with four sockets, each of which had one quad-core processor. Each processor in the socket had an integrated memory controller with a 16 GB DDR3 memory system. Our validation results showed that the models could predict performance and energy consumption within the 11.3% and 13.1% error.

We then used the model to analyze the impact of the TLP, MLP, and memory frequency and used it to search for the optimal configuration. We demonstrated that the model-guided optimization can improve energy consumption up to 40% for applications that exhibit a high demand for memory bandwidth. To conclude, this model provides a new direction for designing new mechanisms that consider the interacting effects among TLP, MLP, and memory frequency to improve both performance and energy for future high-performance computing systems.

6.1.3. Heterogeneous MC design

Lastly, we focused on the memory management of future heterogeneous memory systems. We presented a new memory controller design that combines the best aspects of two baseline heterogeneous memory management policies to manage page resources on heterogeneous memories. We validated our memory controller design in a simulation framework against real hardware on two state-of-the-art HPC servers. We investigated the impact of two policies on performance and energy using HPC workloads, and analyzed the effect of spatial and temporal locality on the energy consumption of both policies. Based on our locality analysis, we proposed a new energy-aware hierarchical memory management policy that dynamically switches between the two policies to optimize energy. The major conclusions and contributions for the three parts of the research are summarized below.

In the third part of the research, we first analyzed two baseline heterogeneous memory organizations and policies, namely PCache and HRank. We argued that neither of these can sustain high performance and low energy consumption across a range of HPC workloads. Thus, we proposed HpMC, a new memory controller design that selectively employs and alternates between PCache and HRank policies to deliver better performance and lower energy consumption. HpMC implements a policy-switching engine (PSE) and several new components that extend a vanilla MC to facilitate switching policies and migrating pages. In addition, HpMC implements an energy-aware controller (EaC). The EaC uses a locality-monitoring engine that periodically analyzes temporal locality based on reuse distance. We defined a temporal locality degree metric used as a guide to switch between PCache and HRank policies, in order to optimize energy consumption. We analyzed the spatial and temporal locality of over 3,000 diverse memory access patterns arising in the Coral Benchmarks and lmbench. We used the results of this experiment to build an energy-optimizing policy switching scheme in the EaC. The results showed that the HpMC reduces energy consumption by 13% to 45% compared to its counterparts, while providing almost the same bandwidth and larger capacity than a DRAM-only system. We concluded that better performance and energy can be achieved via the use of hybrid memory management policies through a well-designed memory controller.

6.2. Future Work

In this dissertation, we proposed a number of methods and technologies to enable resource management and energy-aware computing on emerging heterogeneous, multicore, multi-memory HPC systems. However, there are still many topics that have not been fully explored. Some example research topics are summarized below.

Combined, total system heterogeneity

Reducing power consumption has become critical across all parts of HPC design. HPC venders want improved energy efficiency per computing unit to reduce the total energy cost. Likewise, HPC system managers want to reduce peak power consumption to improve fault tolerance of all system operations. In addition, HPC systems need to keep scaling out the computational ability within a reasonable power budget.

Recent work evaluates and explores the performance and energy consumption for CPU+MIC [170] or CPU+GPU [176] designs. Although these works show promise for heterogeneous system design, the potential of heterogeneous system architectures for HPC systems has not been fully investigated.

We are interested in combining our studies of heterogeneous memories with computationally heterogeneous systems. Our simulation framework (Chapter 5) should provide a foundation for exploring this complicated problem space. This is a major topic for exploration in future work.

Models of combined, total-system performance and energy

To address the power limits of future systems, we need new cost metrics and models to compare the energy efficiency of new system designs. In Chapter 3, we proposed models that can capture essential factors impacting performance and energy efficiency. Although these models can predict performance and energy on a multicore NUMA architecture, it is less clear how they can be extended to predict the performance and energy of different heterogeneous system architectures. Our current models implicitly assume that all hardware resources are homogeneous. In other words, they assume that the computing systems employ the same CPUs and the same memory nodes in the NUMA architecture. In addition, they assume that computing systems only use a flat NUMA memory system. With the introduction of heterogeneous memories, future memory systems may have multiple memory layers. The first layer of memory could be for performance, while the second is for capacity under power constraints. The possibilities are almost endless and models will play a key role in evaluating new designs. Initially, we plan to extend the current models to predict new heterogeneous system designs by adding performance and power parameters that consider heterogeneity.

Resource Management Automation in Emerging Systems

We believe future systems will be heterogeneous in all aspects. This means runtime systems will be required to evaluate the tradeoffs of resource management in an effort to meet user demand efficiently. With future models that consider heterogeneity, we can build sophisticated runtime systems to automatically adapt the programming or system behavior in intelligent ways to carry out an efficient execution. This approach can be leveraged to automatically (1) identify bottlenecks in different system layers, and (2) evaluate the cost of different resource allocations and select the best solution at a fine granularity.

Initially, we can extend the runtime system implemented in Chapter 3, which currently uses DCT and a thread mapping arbiter to allocate thread resources. In the future, the runtime system will incorporate different NUMA data allocation policies to manage data in a heterogeneous memory system and provide a more efficient way to improve thread-and-data affiliation.

This page intentionally left blank.

Bibliography

[1] "*Intel® 64 and IA-32 Architectures Optimization Reference Manual*". Intel Corporation 2009.

[2] "*Micron DDR Power Spreadsheet*". Micron Technology, Inc, 2009.

[3] "*NVIDIA Next Generation CUDA Compute Architecture:Fermi*". Nvidia Inc., 2012.

[4] "*TILEPro64 Processor*". Tilera Inc., 2012.

[5] "*TN-41-01-Calculating Memory System Power for DDR3*". Micron Technology, Inc, 2009.

[6] "*US department of energy annual report*", 2010 http://www.eia.doe.gov/

[7] "*Intel® VTune™ Amplifier XE 2013* ". Intel Inc., 2013 http://software.intel.com/en-us/intel-vtune-amplifier-xe

[8] "*Intel® Many Integrated Core Architecture*", 2012 http://www.intel.com/content/www/us/en/architecture-and-technology/many-integrated-core/intel-many-integrated-core-architecture.html

[9] "*Intel Parallel Advisor*". Intel Corporation, 2014 http://software.intel.com/en-us/articles/intel-parallel-advisor/

[10] "*Tilera TilePro64 Processor*", 2012 http://www.tilera.com/

[11] "*CORAL Benchmarks*". LLNL, 2014 https://asc.llnl.gov/CORAL-benchmarks/

[12] "*512Mb: x4, x8, x16 DDR SDRAM Features*". Micron Technology, Inc, 2007 http://download.micron.com/pdf/datasheets/dram/ddr/512MBDDRx4x8x16.pdf

[13] "*Calculating DDR Memory System Power*", 2007 http://www.micron.com/~/media/Documents/Products/Technical/Note/DRAM/TN4603.pdf

[14] "*TOP500 Supercomputer Site*", http://www.top500.org

[15] "*AMD SimNow Simulator*". AMD 2014 http://developer.amd.com/tools-and-sdks/cpu-development/simnow-simulator/

[16] "*CPUSpeed*". Carl Thompson, 2012 http://www.carlthompson.net/software/cpuspeed

[17] "*Oprofile Performance Monitoring Tool*", 2014 http://oprofile.sourceforge.net/news/

[18] "*Rambus - DRAM Power Model*". Rambus Inc., 2010 www.rambus.com/energy/

[19] "*Calculating Memory System Power for DDR2*". Micron Inc., 2011 http://www.micron.com/~/media/Documents/Products/Technical%20Note/DRAM/tn4704.pdf

[20] "*WattsUp Meter Tool*", 2014 https://www.wattsupmeters.com

[21] Abts, D., Jerger, N. D. E., Kim, J., Gibson, D. and Lipasti, M. H. "Achieving Predictable Performance Through Better Memory Controller Placement in Many-Core CMPs". *SIGARCH Comput. Archit. News*, 37(3), pp. 451-461, 2009.

[22] Aggarwal, N., Cantin, J. F., Lipasti, M. H. and Smith, J. E. "Power-Efficient DRAM Speculation". *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'08)*, 2008.

[23] Ahn, J. H., Jouppi, N. P., Kozyrakis, C., Leverich, J. and Schreiber, R. S. "Future scaling of processor-memory interfaces". *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, (SC'09)*, 2009.

[24] Amdahl, G. M. "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities". *Proceedings of the April 18-20, 1967, spring joint computer conference (AFIPS '67)*, Atlantic City, New Jersey, 1967.

[25] Awasthi, M., Nellans, D. W., Sudan, K., Balasubramonian, R. and Davis, A. "Handling the Problems and Opportunities Posed by Multiple on-Chip Memory Controllers". *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*, Vienna, Austria, 2010.

[26] Azimi, R., Tam, D. K., Soares, L. and Stumm, M. "Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring". *ACM Operating Systems Review (SIGOPS'09)*, pp. 56-65, 2009.

[27] Bae, C. S. "*Dynamic adaptive resource management in a virtualized numa multicore system for optimizing power, energy, and performance*". Northwestern University, 2013.

[28] Bailey, D. H. "Performance and the NAS Parallel Benchmarks". *International Journal of High Performance Computing Applications*, 5(3), pp. 63-73, 1994.

[29] Barbeau, M. and Kranakis, E. "*Principles of ad hoc networking*". Wiley, 2007.

[30] Bardhan, S. and Menascé, D. A. "Analytic Models of Applications in Multi-core Computers". *Proceedings of the 2013 IEEE 21st International Symposium on*

Modelling, Analysis & Simulation of Computer and Telecommunication Systems, 2013.

[31] Bartal, Y., Charikar, M. and Indyk, P. "On Page Migration and Other Relaxed Task Systems". *Theor. Comput. Sci.*, 268(1), pp. 43-66, 2001.

[32] Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hiller, J., Karp, S., Keckler, S., Klein, D., Lucas, R., Richards, M., Scarpelli, A., Scott, S., Snavely, A., Sterling, T., Williams, R. S., Yelick, K., Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hiller, J., Keckler, S., Klein, D., Kogge, P., Williams, R. S. and Yelick, K. "*ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor & Study Lead"*. 2008.

[33] Bianchini, R. and Rajamony, R. "Power and Energy Management for Server Systems". *Computer*, 37(11), pp. 68-76, 2004.

[34] Blagodurov, S., Zhuravlev, S. and Fedorova, A. "Contention-Aware Scheduling on Multicore Systems". *ACM Trans. Comput. Syst.*, 28(4), pp. 1-45, 2010.

[35] Blagodurov, S., Zhuravlev, S., Fedorova, A. and Kamali, A. "A Case for NUMA-Aware Contention Management on Multicore Systems". *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT'10)*, Vienna, Austria, 2010.

[36] Burr, G. W., Kurdi, B. N., Scott, J. C., Lam, C. H., Gopalakrishnan, K. and Shenoy, R. S. "Overview of Candidate Device Technologies for Storage-class Memory". *IBM J. Res. Dev.*, 52(4), pp. 449-464, 2008.

[37] Cameron, K. W., Ge, R. and Feng, X. "High-Performance, Power-Aware Distributed Computing for Scientific Applications". *Computer*, 38(11), pp. 40-47, 2005.

[38] Carrera, E. V., Pinheiro, E. and Bianchini, R. "Conserving Disk Energy in Network Servers". *Proceedings of the 17th ACM International Conference on Supercomputing (ICS'03)*, San Francisco, CA, USA, 2003.

[39] Caulfield, A. M., Coburn, J., Mollov, T., De, A., Akel, A., He, J., Jagatheesan, A., Gupta, R. K., Snavely, A. and Swanson, S. "Understanding the Impact of Emerging Non-Volatile Memories on High-Performance, IO-Intensive Computing". *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis(SC'10)*, 2010.

[40] Chandra, R., Devine, S., Verghese, B., Gupta, A. and Rosenblum, M. "Scheduling and Page Migration for Multiprocessor Compute Servers". *SIGOPS Oper. Syst. Rev.*, 28(5), pp. 12-24, 1994.

[41] Charles, J., Jassi, P., Ananth, N. S., Sadat, A. and Fedorova, A. "Evaluation of the Intel Core i7 Turbo Boost feature". *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC'09)*, 2009.

[42] Chen, E., Apalkov, D., Diao, Z., Driskill-Smith, A., Druist, D., Lottis, D., Nikitin, V., Tang, X., Watts, S., Wang, S., Wolf, S. A., Ghosh, A. W., Lu, J. W., Poon, S. J., Stan, M., Butler, W. H., Gupta, S., Mewes, C. K. A., Mewes, T. and Visscher, P. B. "Advances and Future Prospects of Spin-Transfer Torque Random Access Memory". *IEEE Transactions on Magnetics*, 46(6), pp. 1873-1878, 2010.

[43] Chen, G. and Stenstrom, P. "Critical Lock Analysis: Diagnosing Critical Section Bottlenecks in Multithreaded Applications". *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis(SC'12)*, Salt Lake City, Utah, 2012.

[44] Chen, Z., Lu, Y., Xiao, N. and Liu, F. "A hybrid memory built by SSD and DRAM to support in-memory Big Data analytics". *Knowl. Inf. Syst.*, 41(2), pp. 335-354, 2014.

[45] Cho, S. and Jin, L. "Managing Distributed, Shared L2 Caches through OS-Level Page Allocation". *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'39)*, 2006.

[46] Cho, S. and Lee, H. "Flip-N-Write: A Simple Deterministic Technique to Improve PRAM Write Performance, Energy and Endurance". *Proceedings of the 42th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'42)*, 2009.

[47] Cho, S. and Melhem, R. G. "On the Interplay of Parallelization, Program Performance, and Energy Consumption". *IEEE Trans. Parallel Distrib. Syst.*, 21(3), pp. 342-353, 2010.

[48] Choi, J. W., Bedard, D., Fowler, R. and Vuduc, R. "A Roofline Model of Energy". *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, 2013.

[49] Chua, L. O. "Memristor-The missing circuit element". *IEEE Transactions on Circuit Theory*, 18(5), pp. 507-519, 1971.

[50] Cuppu, V. and Jacob, B. "Concurrency, Latency, or System Overhead: Which Has the Largest Impact on Uniprocessor DRAM-System Performance?". *Proceedings of the 28th annual international symposium on Computer architecture (ISCA'01)*, 2001.

[51] Cuppu, V., Jacob, B., Davis, B. and Mudge, T. "A Performance Comparison of Contemporary DRAM Architectures". *Proceedings of the 26th International Symposium on Computer Architecture (ISCA'99)*, 1999.

[52] Curtis-Maury, M., Blagojevic, F., Antonopoulos, C. D. and Nikolopoulos, D. S. "Prediction-Based Power-Performance Adaptation of Multithreaded Scientific Codes". *IEEE Trans. Parallel Distrib. Syst. (TPDS'08)*, 19(10), pp. 1396-1410, 2008.

[53] Curtis-Maury, M., Dzierwa, J., Antonopoulos, C. D. and Nikolopoulos, D. S. "Online Power-Performance Adaptation of Multithreaded Programs Using Hardware Event-Based Prediction". *Proceedings of the 2005 ACM/IEEE conference on Supercomputing (SC'05)*, 2006.

[54] Curtis-Maury, M., Shah, A., Blagojevic, F., Nikolopoulos, D. S., de Supinski, B. R. and Schulz, M. "Prediction Models for Multi-Dimensional Power-Performance Optimization on Many Cores". *Proceedings of the 17th international conference on Parallel architectures and compilation techniques (PCAT'08)*, 2008.

[55] Curtis-Maury, M., Singh, K., McKee, S. A., Blagojevic, F., Nikolopoulos, D. S., de Supinski, B. R. and Schulz, M. "Identifying Energy-Efficient Concurrency Levels Using Machine Learning". *Proceedings of the 2007 IEEE International Conference on Cluster Computing*, 2007.

[56] D'Auria, B. "A short note on the monotonicity of the Erlang C formula in the Halfin-Whitt regime". *Queueing Syst*, 71(4), pp. 469-472, 2012.

[57] Dashti, M., Fedorova, A., Funston, J., Gaud, F., Lachaize, R., Lepers, B., Quema, V. and Roth, M. "Traffic management: a holistic approach to memory placement on NUMA systems". *SIGARCH Comput. Archit. News*, 41(1), pp. 381-394, 2013.

[58] David, H., Fallin, C., Gorbatov, E., Hanebutte, U. R. and Mutlu, O. "Memory Power Management via Dynamic Voltage/Frequency Scaling". *Proceedings of the 8th ACM international conference on Autonomic computing*, Karlsruhe, Germany, 2011.

[59] David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R. and Le, C. "RAPL: Memory Power Estimation and Capping". *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, 2010.

[60] Delaluz, V., Kandemir, M., Vijaykrishnan, N., Sivasubramaniam, A. and Irwin, M. J. "DRAM Energy Management Using Software and Hardware Directed Power Mode Control". *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'01)*, 2001.

[61] Deng, Q., Meisner, D., Bhattacharjee, A., Wenisch, T. F. and Bianchini, R. "MultiScale: memory system DVFS with multiple memory controllers". *Proceedings of the ACM/IEEE international symposium on Low power electronics and design*, 2012.

[62] Deng, Q., Meisner, D., Ramos, L., Wenisch, T. F. and Bianchini, R. "MemScale: Active Low-Power Modes For Main Memory". *Proceedings of the sixteenth*

international conference on Architectural support for programming languages and operating systems (ASPLOS'11), Newport Beach, California, USA, 2011.

[63] Di Ventra, M., Pershin, Y. V. and Chua, L. O. "Circuit Elements With Memory: Memristors, Memcapacitors, and Meminductors". *Proceedings of the IEEE*, 97(10), pp. 1717-1724, 2009.

[64] Diener, M., Cruz, E. H. M., Navaux, P. O. A., Busse, A., Hei, H.-U. and #223. "kMAF: automatic kernel-level management of thread and data affinity". *Proceedings of the 23rd international conference on Parallel architectures and compilation*, Edmonton, AB, Canada, 2014.

[65] Diniz, B., Guedes, D., Wagner Meira, J. and Bianchini, R. "Limiting the Power Consumption of Main Memory". *SIGARCH Comput. Archit. News*, 35(2), pp. 290-301, 2007.

[66] Dong, X., Xie, Y., Muralimanohar, N. and Jouppi, N. P. "Simple but Effective Heterogeneous Main Memory with On-Chip Memory Controller Support". *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'10)* 2010.

[67] Eyerman, S. and Eeckhout, L. "Modeling Critical Sections in Amdahl's Law and its Implications for Multicore Design". *Proceedings of the 37th annual international symposium on Computer architecture (ISCA'10)*, Saint-Malo, France, 2010.

[68] Fan, X., Ellis, C. and Lebeck, A. "Memory Controller Policies for DRAM Power Management". *Proceedings of the international symposium on Low power electronics and design*, Huntington Beach, California, USA, 2001.

[69] Fedorova, A., Seltzer, M. and Smith, M. D. "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler". *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT'07)*, 2007.

[70] Feng, W.-c. "Making a Case for Efficient Supercomputing". *Queue*, 1(7), pp. 54-64, 2003.

[71] François, B., Nathalie, F., Brice, G., Raymond, N. and Pierre-André, W. "Dynamic Task and Data Placement over NUMA Architectures: An OpenMP Runtime Perspective". *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, 2009.

[72] Freeh, V. W. and Lowenthal, D. K. "Using Multiple Energy Gears in MPI Programs on a Power-Scalable Cluster". *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, Chicago, IL, USA, 2005.

[73] Fu, C., Zhao, M., Xue, C. J. and Orailoglu, A. "Sleep-aware variable partitioning for energy-efficient hybrid PRAM and DRAM main memory". *Proceedings of the 2014 international symposium on Low power electronics and design*, La Jolla, California, USA, 2014.

[74] Gabriel, E., Fagg, G., Bosilca, G., Angskun, T., Dongarra, J., Squyres, J., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R., Daniel, D., Graham, R. and Woodall, T. "*Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation"*. Springer Berlin Heidelberg, 2004.

[75] Ge, R., Feng, X., Burtscher, M. and Zong, Z. "Performance and Energy Modeling for Cooperative Hybrid Computing". *Proceedings of the 2014 9th IEEE International Conference on Networking, Architecture, and Storage*, 2014.

[76] Ge, R., Feng, X. and Cameron, K. W. "Improvement of Power-Performance Efficiency for High-End Computing". *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 11 - Volume 12*, 2005.

[77] Ge, R., Feng, X. and Cameron, K. W. "Modeling and Evaluating Energy-Performance Efficiency of Parallel Processing on Multicore Based Power Aware Systems". *Proceedings of the 23th IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, 2009.

[78] Gross, D., Shortle, J. F., Thompson, J. M. and Harris, C. M. "*Fundamentals of Queueing Theory"*. Wiley-Interscience, 2008.

[79] Guan, N., Stigge, M., Yi, W. and Yu, G. "Cache-Aware Scheduling and Analysis for Multicores". *Proceedings of the seventh ACM international conference on Embedded software*, Grenoble, France, 2009.

[80] Gulur, N. D., Manikantan, R., Mehendale, M. and Govindarajan, R. "Multiple Sub-row Buffers in DRAM: Unlocking Performance and Energy Improvement Opportunities". *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'12)*, 2012.

[81] Gupta, V. and Nathuji, R. "Analyzing Performance Asymmetric Multicore Processors for Latency Sensitive Datacenter Applications". *Proceedings of the 2010 international conference on Power aware computing and systems*, Vancouver, BC, Canada, 2010.

[82] Gustafson, J. L. "Reevaluating Amdahl's law". *Commun. ACM*, 31(5), pp. 532-533, 1988.

[83] Hackenberg, D., Molka, D. and Nagel, W. E. "Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems". *Proceedings of the 42th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'42)*,

New York, New York, 2009.

[84] Hall, M. W. and Martonosi, M. "Adaptive Parallelism in Compiler-Parallelized Code". *Proceedings of THE 2ND SUIF COMPILER WORKSHOP*, 1997.

[85] Halupka, D., Huda, S., Song, W., Sheikholeslami, A., Tsunoda, K., Yoshida, C. and Aoki, M. "Negative-resistance read and write schemes for STT-MRAM in 0.13 CMOS". *Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2010.

[86] Ham, T. J., Chelepalli, B. K., Xue, N. and Lee, B. C. "Disintegrated control for energy-efficient and heterogeneous memory systems". *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013.

[87] Hanumaiah, V., Vrudhula, S. and Chatha, K. S. "Performance Optimal Online DVFS and Task Migration Techniques for Thermally Constrained Multi-Core Processors". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(11), pp. 1677-1690, 2011.

[88] Hashemi, A. H., Kaeli, D. R. and Calder, B. "Efficient Procedure Mapping Using Cache Line Coloring". *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, Las Vegas, Nevada, USA, 1997.

[89] Hay, A., Strauss, K., Sherwood, T., Loh, G. H. and Burger, D. "Preventing PCM Banks from Seizing Too Much Power". *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'44)*, 2011.

[90] Haykin, S. "*Neural Networks -- a Comprehensive Foundation, Prentice Hall"*. Prentice Hall, 1999.

[91] Hill, M. D. and Marty, M. R. "Amdahl's Law in the Multicore Era". *Computer*, 41(7), pp. 33-38, 2008.

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

[93] Hom, J. and Kremer, U. "Inter-Program Optimizations for Conserving Disk Energy". *Proceedings of the 2005 International Symposium on Low Power Electronics and Design, (ISLPED '05)*, 8-10 Aug. 2005.

[94] Hsu, C.-H. and Feng, W.-C. "A Power-Aware Run-Time System for High-Performance Computing". *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, (SC'05)*, 12-18 Nov. 2005.

[95] Hsu, C.-H. and Kremer, U. "The Design, Implementation, and Evaluation of a

Compiler Algorithm for CPU Energy Reduction". *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, San Diego, California, USA, 2003.

[96] Huang, C.-C. and Nagarajan, V. "ATCache: reducing DRAM cache latency via a small SRAM tag cache". *Proceedings of the 23rd international conference on Parallel architectures and compilation*, Edmonton, AB, Canada, 2014.

[97] Huang, H., Pillai, P. and Shin, K. G. "Design and Implementation of Power-Aware Virtual Memory". *Proceedings of the annual conference on USENIX Annual Technical Conference*, San Antonio, Texas, 2003.

[98] Hur, I. and Lin, C. "A Comprehensive Approach to DRAM Power Management". *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'08)*, 2008.

[99] Ipek, E., Mutlu, O., Martinez, J. F. and Caruana, R. "Self-Optimizing Memory Controllers: A Reinforcement Learning Approach". *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*, 2008.

[100] Isci, C., Buyuktosunoglu, A., Chen, C. Y., Bose, P. and Martonosi, M. "An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget". *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'39)*, Dec. 2006, 2006.

[101] Jeon, D., Garcia, S., Louie, C. and Taylor, M. B. "Kismet: Parallel Speedup Estimates for Serial Programs". *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, Portland, Oregon, USA, 2011.

[102] Jevdjic, D., Volos, S. and Falsafi, B. "Die-stacked DRAM caches for servers: hit ratio, latency, or bandwidth? have it all with footprint cache". *Proceedings of the 40th Annual International Symposium on Computer Architecture*, Tel-Aviv, Israel, 2013.

[103] Jiang, Y., Zhang, E. Z., Tian, K. and Shen, X. "Is Reuse Distance Applicable to Data Locality Analysis on Chip Multiprocessors?". *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, 2010.

[104] Jung, C., Lim, D., Lee, J. and Han, S. "Adaptive Execution Techniques for SMT Multiprocessor Architectures". *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, Chicago, IL, USA, 2005.

[105] Justin, M. and Li, J. "*Evaluating Row Buffer Locality in Future Non-Volatile*

Main Memories". 2012.

[106] Karp, A. H. and Flatt, H. P. "Measuring Parallel Processor Performance". *Commun. ACM*, 33(5), pp. 539-543, 1990.

[107] Kendall, D. G. "Stochastic Processes Occurring in the Theory of Queues and their Analysis by the Method of the Imbedded Markov Chain". *The Annals of Mathematical Statistics*, 24(3), pp. 338-354, 1953.

[108] Kgil, T., Roberts, D. and Mudge, T. "Improving NAND Flash Based Disk Caches". *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*, 2008.

[109] Kim, M., P., K., Kim, H. and Brett, B. "Predicting Potential Speedup of Serial Code via Lightweight Profiling and Emulations with Memory Performance Model". *Proceedings of the 26 th IEEE International Parallel and Distributed Processing Symposium (IPDPS'12)*, 21-25 May 2012, 2012.

[110] Lebeck, A. R., Fan, X., Zeng, H. and Ellis, C. "Power Aware Page Allocation". *SIGARCH Comput. Archit. News*, 28(5), pp. 105-116, 2000.

[111] Lee, B. C., Ipek, E., Mutlu, O. and Burger, D. "Architecting Phase Change Memory As a Scalable Dram Alternative". *SIGARCH Comput. Archit. News*, 37(3), pp. 2-13, 2009.

[112] Lee, C. J., Mutlu, O., Narasiman, V. and Patt, Y. N. "Prefetch-Aware DRAM Controllers". *Proceedings of the 41th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'41)*, 2008.

[113] Lee, M., Gupta, V. and Schwan, K. "Software-controlled transparent management of heterogeneous memory resources in virtualized systems". *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, Seattle, Washington, 2013.

[114] Lefurgy, C., Rajamani, K., Rawson, F., Felter, W., Kistler, M. and Keller, T. W. "Energy Management for Commercial Servers". *Computer*, 36(12), pp. 39-48, 2003.

[115] Lefurgy, C., Wang, X. and Ware, M. "Power capping: a prelude to power shifting". *Cluster Computing*, 11(2), pp. 183-195, 2008.

[116] Li, D., de Supinski, B. R., Schulz, M., Cameron, K. and Nikolopoulos, D. S. "Hybrid MPI/OpenMP Power-Aware Computing". *Proceedings of the 23th IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*, 2010.

[117] Li, D., Nikolopoulos, D. S., Cameron, K., de Supinski, B. R. and Schulz, M. "Power-Aware MPI Task Aggregation Prediction for High-End Computing Systems". *Proceedings of the 24th IEEE International Parallel and Distributed Processing*

Symposium (IPDPS'10), 2010.

[118] Li, J. and Martinez, J. F. "Dynamic Power-performance Adaptation of Parallel Computation on Chip Multiprocessors". *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'06)*, 2006.

[119] Li, T., Baumberger, D., Koufaty, D. A. and Hahn, S. "Efficient operating system scheduling for performance-asymmetric multi-core architectures". *Proceedings of the 2007 ACM/IEEE conference on Supercomputing (SC'07)*, Reno, Nevada, 2007.

[120] Li, X., Li, Z., Zhou, Y. and Adve, S. "Performance Directed Energy Management for Main Memory and Disks". *Trans. Storage*, 1(3), pp. 346-380, 2005.

[121] Liedtke, J., Haertig, H. and Hohmuth, M. "OS-Controlled Cache Predictability for Real-Time Systems". *Proceedings of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS '97)*, 1997.

[122] Lim, M. Y., Freeh, V. W. and Lowenthal, D. K. "Adaptive, Transparent Frequency and Voltage Scaling of Communication Phases in MPI Programs". *Proceedings of the 2006 ACM/IEEE conference on Supercomputing (SC'06)*, Tampa, Florida, 2006.

[123] Lin, J., Lu, Q., Ding, X., Zhang, Z., Zhang, X. and P., S. "Gaining Insights into Multicore Cache Partitioning: Bridging the Gap Between Simulation and Real Systems". *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'08)*, 2008.

[124] Lin, J., Zheng, H., Zhu, Z., David, H. and Zhang, Z. "Thermal Modeling and Management of DRAM Memory Systems". *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*, San Diego, California, USA, 2007.

[125] Lin, J., Zheng, H., Zhu, Z., Gorbatov, E., David, H. and Zhang, Z. "Software Thermal Management of DRAM memory for multicore Systems". *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, Annapolis, MD, USA, 2008.

[126] Lin, W.-F., Reinhardt, S. K. and Burger, D. "Designing a Modern Memory Hierarchy with Hardware Prefetching". *IEEE Trans. Comput.*, 50(11), pp. 1202-1218, 2001.

[127] Loh, G. H. "3D-Stacked Memory Architectures for Multi-core Processors". *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*, 2008.

[128] Loh, G. H. and Hill, M. D. "Efficiently enabling conventional block sizes for very large die-stacked DRAM caches". *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, Porto Alegre, Brazil, 2011.

[129] Majo, Z. and Gross, T. R. "Memory Management in NUMA Multicore Systems: Trapped between Cache Contention and Interconnect Overhead". *Proceedings of the International Symposium on Memory Management*, 2011.

[130] Majo, Z. and Gross, T. R. "Memory system performance in a NUMA multicore multiprocessor". *Proceedings of the 4th Annual International Conference on Systems and Storage*, 2011.

[131] Mandal, A., Fowler, R. and Porterfield, A. "Modeling Memory Concurrency for Multi-Socket Multi-Core Systems". *Performance Analysis of Systems & Software (ISPASS)*, 2010.

[132] Marathe, J. and Mueller, F. "Hardware Profile-Guided Automatic Page Placement for ccNUMA Systems". *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, New York, USA, 2006.

[133] Marathe, J., Thakkar, V. and Mueller, F. "Feedback-Directed Page Placement for ccNUMA via Hardware-Generated Memory Traces". *Journal of Parallel and Distributed Computing*, 70(12), pp. 1204-1219, 2010.

[134] Marchetti, M., Kontothanassis, L., Bianchini, R. and Scott, M. L. "Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems". *Proceedings of 9th International Parallel Processing Symposium* 1995.

[135] Marowka, A. "Extending Amdahl's Law for Heterogeneous Computing". *Parallel and Distributed Processing with Applications (ISPA)*, 2012.

[136] Marowka, A. "Maximizing energy saving of dual-architecture processors using DVFS". *J. Supercomput.*, 68(3), pp. 1163-1183, 2014.

[137] McCalpin, J. "*STREAM: Sustainable Memory Bandwidth in High Performance Computers"*. 1995.

[138] McCalpin, J. D. "Memory Bandwidth and Machine Balance in Current High Performance Computers". *IEEE Technical Committee on Computer Architecture (TCCA) Newsletter*, pp., 1995.

[139] McCurdy, C. and Vetter, J. S. "Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms". *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2010.

[140] McKee, S. A. "Reflections on the Memory Wall". *Proceedings of the 1st conference on Computing frontiers*, Ischia, Italy, 2004.

[141] McVoy, L. and Staelin, C. "Lmbench: Portable Tools for Performance Analysis". *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, 1996.

[142] Meza, J., Chang, J., Yoon, H., Mutlu, O. and Ranganathan, P. "Enabling Efficient and Scalable Hybrid Memories Using Fine-Granularity DRAM Cache Management". *IEEE Comput. Archit. Lett.*, 11(2), pp. 61-64, 2012.

[143] Mizell, D. and Maschhoff, K. "Early Experiences with Large-Scale Cray XMT Systems". *Proceedings of the 23th IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*
2009.

[144] Mogul, J. C., Argollo, E., Shah, M. and Faraboschi, P. "Operating System Support for NVM+DRAM Hybrid Main Memory". *Proceedings of the 12th Conference on Hot Topics in Operating Systems*, 2009.

[145] Molka, D., Hackenberg, D., Schone, R. and Muller, M. S. "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System". *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT'09)*, 2009.

[146] Mucci, P. J., Browne, S., Deane, C. and Ho, G. "PAPI: A Portable Interface to Hardware Performance Counters". *Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999.

[147] Mutlu, O. and Moscibroda, T. "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems". *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA'08)*, 2008.

[148] Mutlu, O. and Moscibroda, T. "Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors". *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'40)*, 2007.

[149] Nikolopoulos, D. S., Papatheodorou, T. S., Polychronopoulos, C. D., Labarta, J. and Ayguad, E. "Leveraging Transparent Data Distribution in OpenMP via User-Level Dynamic Page Migration". *Proceedings of the Third International Symposium on High Performance Computing*, 2000.

[150] Nikolopoulos, D. S., Papatheodorou, T. S., Polychronopoulos, C. D., Labarta, J. and Ayguade, E. "User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors". *International Conference on Parallel Processing(ICPP'00)*, 2000.

[151] Nikolopoulos, D. S., Papatheodorou, T. S., Polychronopoulos, C. D., Labarta, J. and Ayguadé, E. "UPMLIB: A Runtime System for Tuning the Memory Performance of OpenMP Programs on Scalable Shared-Memory Multiprocessors". *Proceedings of the 5th International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, 2000.

[152] Pawlowski, J. T. "*Hybrid memory cube (HMC)"*. 2011.

[153] Quan, C. "WATS: Workload-Aware Task Scheduling in Asymmetric Multi-core Architectures". *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS'12)*, 2012.

[154] Qureshi, M. K., Franceschini, M. M., Jagmohan, A. and Lastras, L. A. "PreSET: Improving performance of phase change memories by exploiting asymmetry in write times". *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA'12).*

[155] Qureshi, M. K., Franceschini, M. M., Lastras-Montaño, L. A. and Karidis, J. P. "Morphable Memory System: A Robust Architecture for Exploiting Multi-level Phase Change Memories". *SIGARCH Comput. Archit. News*, 38(3), pp. 153-162, 2010.

[156] Qureshi, M. K., Karidis, J., Franceschini, M., Srinivasan, V., Lastras, L. and Abali, B. "Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling". *Proceedings of the 42th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'42)*, 2009.

[157] Qureshi, M. K. and Loh, G. H. "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design". *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, Vancouver, B.C., CANADA, 2012.

[158] Qureshi, M. K., Srinivasan, V. and Rivers, J. A. "Scalable High Performance Main Memory System Using Phase-Change Memory Technology". *Proceedings of the 36th annual international symposium on Computer architecture (ISCA'09)*, Austin, TX, USA, 2009.

[159] Ramos, L. E., Gorbatov, E. and Bianchini, R. "Page Placement in Hybrid Memory Systems". *Proceedings of the 26th ACM International Conference on Supercomputing (ICS'11)*, Tucson, Arizona, USA, 2011.

[160] Ribeiro, C. P., Mehaut, J. F., Carissimi, A., Castro, M. and Fernandes, L. G. "Memory Affinity for Hierarchical Shared Memory Multiprocessors". *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on*, 2009/oct.

[161] Rodrigues, R., Annamalai, A., Koren, I. and Kundu, S. "Scalable Thread Scheduling in Asymmetric Multicores for Power Efficiency". *Proceedings of IEEE*

24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'12), 2012.

[162] Rong, G., Xizhou, F. and Cameron, K. W. "Modeling and evaluating energy-performance efficiency of parallel processing on multicore based power aware systems". *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 23-29 May 2009, 2009.

[163] Rong, G., Xizhou, F., Wu-chun, F. and Cameron, K. W. "CPU MISER: A Performance-Directed, Run-Time System for Power-Aware Clusters". *International Conference on Parallel Processing(ICPP'07)*
2007.

[164] Rosenfeld, P., Cooper-Balis, E. and Jacob, B. "DRAMSim2: A Cycle Accurate Memory System Simulator". *Computer Architecture Letters*, 10(1), pp. 16-19, 2011.

[165] Saripalli, V., Guangyu, S., Mishra, A., Yuan, X., Datta, S. and Narayanan, V. "Exploiting Heterogeneity for Energy Efficiency in Chip Multiprocessors". *Emerging and Selected Topics in Circuits and Systems, IEEE Journal on*, 1(2), pp. 109-119, 2011.

[166] Sawalha, L. and Barnes, R. D. "Phase-Based Scheduling and Thread Migration for Heterogeneous Multicore Processors". *Proceedings of the 21th International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*, Minneapolis, Minnesota, USA, 2012.

[167] Sawalha, L. and Barnes, R. D. "Energy-Efficient Phase-Aware Scheduling for Heterogeneous Multicore Processors". *Green Technologies Conference, 2012 IEEE*, 2012.

[168] Scheurich, C. and Dubois, M. "Dynamic Page Migration in Multiprocessors with Distributed Global Memory". *IEEE Transactions on Computers*, 38(8), pp. 1154-1163, 1989.

[169] Schuff, D. L., Kulkarni, M. and Pai, V. S. "Accelerating Multicore Reuse Distance Analysis with Sampling and Parallelization". *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*, 2010.

[170] Shao, Y. S. and Brooks, D. "Energy characterization and instruction-level energy model of Intel's Xeon Phi processor". *Low Power Electronics and Design (ISLPED)*, 2013.

[171] Shen, X., Shaw, J., Meeker, B. and Ding, C. "Locality Approximation Using Time". *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.

[172] Sheu, S.-S., Chang, M.-F., Lin, K.-F., Wu, C.-W., Chen, Y.-S., Chiu, P.-F., Kuo, C.-C., Yang, Y.-S., Chiang, P.-C., Lin, W.-P., Lin, C.-H., Lee, H.-Y., Gu, P.-Y., Wang, S.-M., Chen, F. T., Su, K.-L., Lien, C.-H., Cheng, K.-H., Wu, H.-T., Ku, T.-K., Kao, M.-J. and Tsai, M.-J. "A 4Mb embedded SLC resistive-RAM macro with 7.2ns read-write random-access time and 160ns MLC-access capability". *Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2011.

[173] Shuaiwen, S., Chun-Yi, S., Rong, G., Vishnu, A. and Cameron, K. W. "Iso-Energy-Efficiency: An Approach to Power-Constrained Parallel Computation". *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11)*, 2011.

[174] Singh, K., Curtis-Maury, M., McKee, S. A., Blagojević, F., Nikolopoulos, D. S., de Supinski, B. R. and Schulz, M. "Comparing Scalability Prediction Strategies on an SMP of CMPs". *Proceedings of the 16th international Euro-Par conference on Parallel processing*, 2010.

[175] Snavely, A. and Tullsen, D. M. "Symbiotic Jobscheduling for a Simultaneous Multithreaded Processor". *SIGARCH Comput. Archit. News*, 28(5), pp. 234-244, 2000.

[176] Song, S., Su, C., Rountree, B. and Cameron, K. W. "A Simplified and Accurate Model of Power-Performance Efficiency on Emergent GPU Architectures". *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS'13)*, 2013.

[177] Stonebraker, M., Frew, J., Gardels, K. and Meredith, J. "The Sequoia 2000 Benchmark". *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 1993.

[178] Su, C., Li, D., Nikolopoulos, D. S., Cameron, K. W., Supinski, B. R. d. and Leon, E. A. "Model-based, memory-centric performance and power optimization on NUMA multiprocessors". *Proceedings of the 2012 IEEE International Symposium on Workload Characterization (IISWC'12)*, 2012.

[179] Su, C., Li, D., Nikolopoulos, D. S., Grove, M., Cameron, K. and Supinski, B. R. d. "Critical path-based thread placement for NUMA systems". *SIGMETRICS Perform. Eval. Rev.*, 40(2), pp. 106-112, 2012.

[180] Suleman, M. A., Qureshi, M. K. and Patt, Y. N. "Feedback-Driven Threading: Power-Efficient and High-Performance Execution of Multi-Threaded Workloads on CMPs". *SIGOPS Oper. Syst. Rev.*, 42(2), pp. 277-286, 2008.

[181] Sun, X.-H. and Ni, L. M. "Another View on Parallel Speedup". *Proceedings of the 1990 ACM/IEEE conference on Supercomputing (SC'90)*, New York, New York, USA, 1990.

[182] Sun, X.-H. and Ni, L. M. "Scalable Problems and Memory-Bounded Speedup". *J. Parallel Distrib. Comput.*, 19(1), pp. 27-37, 1993.

[183] Taecheol, O., Hyunjin, L., Kiyeon, L. and Sangyeun, C. "An Analytical Model to Study Optimal Area Breakdown between Cores and Caches in a Chip Multiprocessor". *IEEE Computer Society Annual Symposium on VLSI*, 2009.

[184] Tam, D., Azimi, R. and Stumm, M. "Thread Clustering: Sharing-Aware Scheduling on SMP-CMP-SMT Multiprocessors". *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems* 2007.

[185] Tam, D. K., Azimi, R., Soares, L. B. and Stumm, M. "RapidMRC: Approximating L2 Miss Rate Curves on Commodity Systems for Online Optimizations". *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, Washington, DC, USA, 2009.

[186] Terboven, C., an Mey, D., Schmidl, D., Jin, H. and Reichstein, T. "Data and Thread Affinity in OpenMP Programs". *Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?*, 2008.

[187] Tolentino, M. and Cameron, K. W. "The Optimist, the Pessimist, and the Global Race to Exascale in 20 Megawatts". *Computer*, 45(1), pp. 95-97, 2012.

[188] Tolentino, M. E., Turner, J. and Cameron, K. W. "Memory-MISER: A Prformance-Constrained Runtime System for Power-Scalable Clusters". *Proceedings of the 4th international conference on Computing frontiers*, Ischia, Italy, 2007.

[189] Tomov, S., Dongarra, J. and Baboulin, M. "Towards dense linear algebra for hybrid GPU accelerated manycore systems". *Parallel Comput.*, 36(5-6), pp. 232-240, 2010.

[190] Tong, L., Brett, P., Knauerhase, R., Koufaty, D., Reddy, D. and Hahn, S. "Operating System Support for Overlapping-ISA Heterogeneous Multi-Core Architectures". *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'10)*, 2010.

[191] Treibig, J., Hager, G. and Wellein, G. "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments". *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, 2010.

[192] Vantrease, D., Schreiber, R., Monchiero, M., McLaren, M., Jouppi, N. P., Fiorentino, M., Davis, A., Binkert, N., Beausoleil, R. G. and Ahn, J. H. "Corona: System Implications of Emerging Nanophotonic Technology". *SIGARCH Comput. Archit. News*, 36(3), pp. 153-164, 2008.

[193] Verghese, B., Devine, S., Gupta, A. and Rosenblum, M. "Operating System Support for Improving Data Locality on CC-NUMA Compute Servers". *SIGOPS Oper. Syst. Rev.*, 30(5), pp. 279-289, 1996.

[194] Vivek, P., Jiang, W., Yuanyuan, Z. and Bianchini, R. "DMA-Aware Memory Energy Management". *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'06)*, 2006.

[195] Voss, M. and Eigenmann, R. "Reducing Parallel Overheads through Dynamic Serialization". *Proceedings of 13th International and 10th Symposium on Parallel and Distributed Processing, IPSS*, 1999.

[196] Wang, B., Wu, B., Li, D., Shen, X., Yu, W., Jiao, Y. and Vetter, J. S. "Exploring Hybrid Memory for GPU Energy Efficiency Through Software-hardware Co-design". *Proceedings of the 22th International Conference on Parallel Architectures and Compilation Techniques (PACT'13)*, 2013.

[197] Weinberg, J., McCracken, M. O., Strohmaier, E. and Snavely, A. "Quantifying Locality In The Memory Access Patterns of HPC Applications". *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing(SC'05)*, 2005.

[198] Wentzlaff, D., Griffin, P., Hoffmann, H., Bao, L., Edwards, B., Ramey, C., Mattina, M., Miao, C.-C., III, J. F. B. and Agarwal, A. "On-Chip Interconnection Architecture of the Tile Processor". *IEEE Micro*, 27(5), pp. 15-31, 2007.

[199] Wong, H. S. P., Raoux, S., Kim, S., Liang, J., Reifenberg, J. P., Rajendran, B., Asheghi, M. and Goodson, K. E. "Phase Change Memory". *Proceedings of the IEEE*, 98(12), pp. 2201-2227, 2010.

[200] Wonyoung, K., Gupta, M. S., Gu-Yeon, W. and Brooks, D. "System Level Analysis of Fast, per-core DVFS Using on-Chip Switching Regulators". *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'08)*, 2008.

[201] Wu, M. and Zwaenepoel, W. "eNVy: A Non-volatile, Main Memory Storage System". *SIGPLAN Not.*, 29(11), pp. 86-97, 1994.

[202] Wu, Q., Martonosi, M., Clark, D. W., Reddi, V. J., Connors, D., Wu, Y., Lee, J. and Brooks, D. "Dynamic-Compiler-Driven Control for Microprocessor Energy and Performance". *Micro, IEEE*, 26(1), pp. 119-129, 2006.

[203] Xiaowen, C., Zhonghai, L., Jantsch, A. and Shuming, C. "Speedup Analysis of Data-Parallel Applications on Multi-core NoCs". *ASIC. ASICON '09. IEEE 8th International Conference on*, 2009.

[204] Yang, B.-D., Lee, J.-E., Kim, J.-S., Cho, J., Lee, S.-Y. and Yu, B.-G. "A Low Power Phase-Change Random Access Memory using a Data-Comparison Write

Scheme". *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*, 2007.

[205] Yang, R., Antony, J., Janes, P. P. and Rendell, A. P. "Memory and Thread Placement Effects as a Function of Cache Usage: A Study of the Gaussian Chemistry Code on the SunFire X4600 M2". *Parallel Architectures, Algorithms, and Networks, 2008. I-SPAN 2008. International Symposium on*, 2008.

[206] Yao, E., Bao, Y., Tan, G. and Chen, M. "Extending Amdahl's Law in the Multicore Era". *SIGMETRICS Perform. Eval. Rev.*, 37(2), pp. 24-26, 2009.

[207] Yongsoo, J., Yongseok, C. and Hojun, S. "Energy Exploration And reduction of SDRAM Memory Systems". *Proceedings of 39th Design Automation Conference*, 2002.

[208] Yoongu, K., Dongsu, H., Mutlu, O. and Harchol-Balter, M. "ATLAS: A Scalable and High-Performance Scheduling Algorithm for Multiple Memory Controllers". *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA'10)*, 2010.

[209] Zhang, W. and Li, T. "Exploring Phase Change Memory and 3D Die-Stacking for Power/Thermal Friendly, Fast and Durable Memory Architectures". *Proceedings of the 18th international conference on Parallel architectures and compilation techniques (PCAT'09)*.

[210] Zhang, X., Dwarkadas, S. and Shen, K. "Towards Practical Page Coloring-Based Multicore Cache Management". *Proceedings of the 4th ACM European conference on Computer systems*, Nuremberg, Germany, 2009.

[211] Zhang, Y., Burcea, M., Cheng, V., Ho, R. and Voss, M. "An Adaptive OpenMP Loop Scheduler for Hyperthreaded SMPs". *Proceedings of International Conference on Parallel and Distributed Computing Systems*, 2004.

[212] Zhang, Y. and Voss, M. "Runtime Empirical Selection of Loop Schedulers on Hyperthreaded SMPs". *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers - Volume 01*, 2005.

[213] Zheng, H., Lin, J., Zhang, Z., Gorbatov, E., David, H. and Zhu, Z. "Mini-rank: Adaptive DRAM Architecture for Improving Memory Power Efficiency". *Proceedings of the 41th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'41)*, 2008.

[214] Zheng, H., Lin, J., Zhang, Z. and Zhu, Z. "Decoupled DIMM: building high-bandwidth memory system using low-speed DRAM devices". *Proceedings of the 37th annual international symposium on Computer architecture (ISCA'10)*, 2009.

[215] Zheng, H. and Zhu, Z. "Power and Performance Trade-Offs in Contemporary

DRAM System Designs for Multicore Processors". *IEEE Transactions on Computers*, 59(8), pp. 1033-1046, 2010.

[216] Zhou, P., Zhao, B., Yang, J. and Zhang, Y. "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology". *SIGARCH Comput. Archit. News*, 37(3), pp. 14-23, 2009.

[217] Zhou, Y., Philbin, J. and Li, K. "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches". *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, 2001.

[218] Zhu, Q., Chen, Z., Tan, L., Zhou, Y., Keeton, K. and Wilkes, J. "Hibernator: Helping Disk arrays Sleep through the Winter". *Proceedings of the twentieth ACM symposium on Operating systems principles*, Brighton, United Kingdom, 2005.

[219] Zhuravlev, S., Blagodurov, S. and Fedorova, A. "Addressing Shared Resource Contention in Multicore Processors via Scheduling". *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.