THREAD SCHEDULING FOR CHIP MULTIPROCESSORS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Major Balram Bhadauria

August 2009

THREAD SCHEDULING FOR CHIP MULTIPROCESSORS

Major Balram Bhadauria, Ph.D.

Cornell University 2009

Large, high frequency single-core chip designs are increasingly being replaced with larger chip multiprocessor (CMP) designs that tradeoff frequency for greater numbers of cores. Power has become a first-order design constraint, leading to designs optimized for computing efficiency, defined as the product of energy and delay (ED). This will continue, based on current technology trends, due to physical thermal and power constraints. To efficiently leverage these new and future generations of hardware, software designers write multithreaded programs. However, creating multithreaded software is non-trivial, and has been an active area of research for several decades. Software developers make certain assumptions of the underlying substrate during product design that may not be true, which can severely affect application performance. For example, software and the operating system assume off-core resources increase in tandem with numbers of cores, which is often not the case. The result is poor performance because scheduling decisions fail to properly account for this non-uniform substrate.

We investigate how to schedule applications for current and future systems when their performance can be limited by frequency heterogeneity among cores, or by the sharing of unified resources. We examine application scaling on both homogeneous and frequency heterogeneous CMPs, and find multithreaded applications often do not scale with increasing numbers of cores, due to off-chip memory limitations and increased contention among threads. Increasing static and off-chip system power consumption can also mask which concurrency level (number of threads) is the most energy efficient for multithreaded programs. Operating system (OS) schedulers are unaware of these is-

sues, and cannot create optimal schedules. We demonstrate how the OS and user-space programs can be made aware of these issues at run-time using hardware performance counters that already exist on chip. We evaluate user-space run-time energy-aware co-scheduling heuristics for running poorly scaling programs at efficient concurrency levels to improve overall performance and energy consumption. Our online scheduling algorithms automate the process of choosing programs to co-schedule, as well as the process of choosing the numbers of threads. Our software schedulers ensure the fairness of co-scheduled applications, and improve computing efficiency for the entire multiprogrammed workload. By using software schedulers, we do not require hardware modifications, making our work portable to different platforms.

We extend this work by investigating thread scheduling and application co-scheduling for frequency heterogenous processors. By extending our schedulers to be aware of processor heterogeneity and mapping application threads to the processors that run them the most efficiently, we achieve improved computing efficiency over scheduling in a processor-oblivious manner.

BIOGRAPHICAL SKETCH

Major is from Toronto, Canada. He received his Bachelor of Applied Science degree in Electrical and Computer Engineering, specializing in Computer Hardware, from the University of Toronto in May 2004. He received his Master of Science degree in Electrical and Computer Engineering from Cornell University in August 2008. Before coming to Cornell University, he worked for two years at ATI Technologies in Markham, Canada and at IC Nexus in Taipei, Taiwan as an ASIC Design Engineer.

This thesis is dedicated to my parents, Dr. Pushpa (Panwar) Bhadauria and Mr. Jagdish Bhadauria, who raised me with lots of love and support and taught me the priceless value of a great education. They are the best role models I could ever have.

I thank my mom for always being there for me and for understanding my potential, even when I didn't always recognize it. I thank my dad for demonstrating how to persistently dream big and aim high, regardless of what others think. Without their support and the support of my siblings, I wouldn't have accomplished so much.

"Money comes and goes, but your education, learning, and accomplishments, are yours to keep forever. No one can take it away from you." My mother, Dr. Pushpa Bhadauria, (said throughout my growing years), which she learned from her parents, which in turn was passed down from their parents.

ACKNOWLEDGMENTS

I would like to thank several individuals and institutions for their help. I acknowledge my advisor, Sally McKee for all the wisdom and knowledge she has given me. I also appreciate the suggestions and insight Karan Singh, Vince Weaver and Brian White have shared with me over the last few years. I thank Intel for donating equipment, and Vince Weaver for maintaining the computing resources that made my research possible. I would also like to acknowledge the NSERC Doctoral Fellowship (PGS-D) I received from the Government of Canada, which has partly funded my research. Lastly, I would like to thank everyone at Cornell's Computer Systems Lab for being such excellent sources of technical knowledge.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

PROBLEMS AND PROPOSED SOLUTIONS

Power and thermal constraints limit the maximum operating frequency of high performance processors, since increasing frequency requires higher voltages and cubic scaling in power. As processor frequencies no longer increase at historical rates, architects have focused on placing more processor cores on-chip to deliver performance improvements via increased parallelism and throughput. This has led to a proliferation of single chip multiprocessors (CMPs) with multiple levels of cache [34]. Multithreaded programs leverage such shared-memory architectures by partitioning workloads and distributing them among cores. However, memory speeds greatly lag behind their processor counterparts, and CMPs do not improve memory latency or bandwidth over single processor chips. In fact, observed latencies may increase due to cache coherence checks and the competition of cores for shared bandwidth and memory resources. Pin limitations limit bandwidth per core.

CMPs can contain multiple sources of contention due to processors sharing multiple chip resources, such as caches, off-chip bandwidth, and system memory. CMPs can also be composed of processors running at varying frequencies, resulting in non-uniform thread performance. Due to these variations, we examine the role of process scheduling in improving computing efficiency. OS schedulers can deliver poor performance on such systems for several reasons. While the OS is aware of the numbers of processors within a system, it is often not aware of CMP non-uniformity. Shared caches, off-chip bandwidth, and memory channels fail to increase in tandem with increasing cores, even though increasing cores results in an increased amount of private cache. The operating system is also not energy-aware and cannot make decisions based on a program's power consumption. Since thermal and power demands limit achievable frequency, power is

a first-order design constraint which must be accounted for when making scheduling decisions. We address these issues by developing real-time heterogeneity and energy-aware schedulers, and test and verify our work on multicore shared memory systems.

Our schedulers achieve performance and energy improvements, both of which are important in the current cost and energy conscious climate for server farms as well as mobile devices. Improving performance saves time, and can reduce costs if computing cycles consumed are billed by a service provider, such as a cloud computing service. Reducing energy also saves power and (indirectly) cooling costs. By co-scheduling, we improve computing efficiency by a factor of 1.5 for a frequency-homogeneous CMP, and by 1.3 for a frequency-heterogeneous CMP. Unlike prior work that examined scheduling for single-core on-chip systems, we target scheduling for CMPs and model contention for resources among threads. CMPs are different from single-core chips because several hardware structures are shared across cores, which can result in contention and reduce application scaling (this is shown in Chapter 2). CMPs also show different power characteristics from single-core single-socket systems, since power consumption does not increase linearly with processors. Our co-schedulers account for and leverage this non-linearity to achieve higher computing efficiency than previous state-of-the-art schedulers [13], [56]. Our work is also important because it targets scheduling of multithreaded programs, which are of increasing importance in the current era of limited frequencies and increasing numbers of processors on-chip. Multithreaded programs are challenging because performance is dependent on the slowest thread, which can be adversely affected if delay is introduced to any thread due to thread synchronization. Executing multiple threads introduces variables that make scheduling solutions from the single-threaded domain obsolete or incompatible. We show current memory bottlenecks are exacerbated with increasing numbers of threads, and on-chip co-scheduling is a viable solution for mitigating the current performance gap between processors and

memory.

We first examine multithreaded program behavior on CMPs and the role that multi-dimensional DRAM structures play in program performance in Chapter 2. This analysis of thread contention motivates us to schedule multiprogrammed workloads consisting of multithreaded programs in Chapter 3 to mitigate these bottlenecks. To broaden the applicability of our work, we consider co-scheduling for heterogeneous processors, in addition to homogenous processors. This motivates us to investigate workload-balancing techniques for running a single multithreaded program on a frequency heterogenous CMP in Chapter 4. We leverage our work in Chapters 3 and 4 to devise application co-scheduling techniques for a frequency heterogeneous CMP in Chapter 5.

1.1 Role of Variable Memory Latency on Multithreaded Program Performance Scaling

The current software paradigm is based on the assumption that multithreaded programs with little contention for shared data scale (nearly) linearly with the numbers of processors, yielding power-efficient data throughput. However, even when each application thread enjoys its own cache and private working set, the application may still fail to achieve maximum throughput [16]. Bandwidth and memory bottlenecks can limit the potential of multicore processors, just as in the single-core domain [11], and in fact these effects are exacerbated by the contention for resources shared across cores. However, we investigate and show that duplicating cache resources along with the processor is not a sufficient condition for application performance to scale with increased numbers of threads.

To explore memory effects of a CMP running multithreaded workloads, we model

a memory controller that uses state-of-the-art scheduling techniques to hide DRAM access latency. Pin limitations affect available memory channels and bandwidth, which can cause applications to scale poorly with increasing numbers of cores. Memory speeds limit the maximum numbers of cores that should be used to achieve the best ratio between power efficiency and performance. Main memory requests are rarely modeled with variable latency, and are instead assumed to have a fixed latency [28]. We compare differences in performance between modeling a static memory latency and modeling a memory controller with realistic, variable DRAM latencies for the SPLASH-2 benchmark suite [67]. We specifically examine CMP efficiency.

Results demonstrate the insufficiency of assuming static latencies: the behaviors caused by such modeling choices not only affect performance but the shapes of the performance curves with respect to numbers of threads. Additionally, we find that choosing the optimal configuration instead of that with the most threads can result in significantly higher computing efficiency. This is a direct result of variable DRAM latencies and the bottlenecks they cause for multithreaded programs. Accurately modeling the multi-dimensional structure of the DRAM exposes the bottleneck at a lower thread count than in earlier research [67], which assumes static latencies.

The contributions of this investigation are fourfold:

1. We find significant non-linear performance differences between bandwidth-limited static latency simulations and those using a realistic DRAM model and memory controller.
2. We find that the maximum numbers of threads is rarely the optimal number to use for performance or power when memory is faithfully modeled.
3. We present a cycle-accurate command-based DRAM DDR2 memory model incorporated within a CMP simulator, which can be used by others to further their

own research.

4. We extend the DDR2 memory model to include state of the art memory optimizations commonly found in today's commercial processors.

The results of our investigation show how thread contention leads to limited or negative performance gains with increasing threads. This scaling behavior manifests itself on CMPs because bandwidth, DRAM banks, and the memory controller are shared among multiple cores.

1.2 Energy and Performance Aware Co-Scheduling for CMPs

Multithreaded programs leverage shared-memory architectures by partitioning workloads and distributing them among different virtual threads, which are then allocated to available cores. However, improvements in memory speeds greatly lag behind their processor counterparts, and observed memory latencies may increase over uniprocessor chips due to cache coherence checks. This can result in suboptimal program scaling with increasing thread counts.

Given the processor-memory performance gap, cores (especially in high-performance applications) are often bottlenecked by memory speeds [50]. This gap widens with increasing cores (due to greater pressure on off-chip communication [6]). Current high-performance scientific computing benchmarks exhibit sub-linear gains with number of threads due to memory constraints. The standard convention has been to use the most threads possible, however, this may not be the most energy- or performance-efficient solution for a multiprogrammed workload of multithreaded programs. Devoting the entire CMP to each program in a workload might deliver best

performance, but that is infeasible unless as many CMPs as applications are available. Otherwise, the computing resources need to be time-shared among applications.

There are three main types of job scheduling: time sharing, space sharing, and space-time sharing. Many high-performance scientific and commercial workloads that run on shared-memory systems use time-sharing of programs, gang-scheduling their respective threads (in an effort to obtain best performance from less thrashing and fewer conflicts for shared resources). This scheduling policy thus provides the baseline for previous studies [6, 56, 13]. In contrast, we discover that better performance for several multi-threaded programs results from space sharing rather than time-sharing the CMP among applications. If cloud computing services are used, where customers are billed based on the number of compute cycles consumed, running programs in isolation can cost more and use more energy than scheduling them together.

We derive efficient runtime program schedules based on memory behavior for cases where a lack of processors requires time sharing among programs. In particular, we use performance counters to identify when programs fail to scale. We explore several local-search heuristics for scheduling multithreaded programs concurrently and for choosing the number of threads per program. Our approach improves use of computing resources and reduces energy costs. We make several contributions:

1. We introduce the concept of power-aware thread scheduling for multithreaded programs.
2. We introduce a space-sharing algorithm that allocates processors in a holistic manner based on resource consumption, without user input or code recompilation.
3. We introduce processor allocation based on the gain and loss of neighbor applications.

4. We account for changing cache behavior with concurrency levels through real-time monitoring of performance.
5. We find co-scheduling more than two programs increases energy efficiency.
6. We verify our energy and performance benefits via real hardware.

Our co-scheduler takes a holistic, synergistic (HOLISYN) approach to scheduling: it accounts for performance and energy when co-scheduling to achieve better energy and performance than running applications in isolation. We improve total CMP throughput by balancing hardware resource requirements of individual applications. Applications selected to run concurrently complement one another so as to reduce the probability of resource conflicts. The advantage of our approach is we can infer software behavior from the hardware using performance counters, without requiring user knowledge or recompilation. Our systematic methodology asymptotically iterates to find an efficient configuration without having to sample the entire design space of application thread counts and programs that can be co-scheduled. Since we determine performance at runtime, we can revert back to time sharing the CMP, if co-scheduling reduces performance. We look at the power and energy advantages of adaptive co-scheduling, as well as space sharing based on an application's resource consumption. We test and verify our work on real hardware CMPs, chiefly an eight-core SMP system composed of several dual-core CMPs. We use the PARSEC benchmark suite [8] to illustrate scheduling benefits, since these programs are designed for a multithreaded environment. We focus on multithreaded programs because they are of increasing importance for achieving good performance in a multicore era. For poorly scaling programs, we deliver significant performance improvements over time-sharing and resource oblivious space-sharing methods. Our schedulers improve overall performance by 19%, and reduce energy consumption by 26%.

1.3 Workload Balancing for CMPs with Heterogeneous Frequencies

With single core chips, CPU manufacturers detect under-performing processors at production time and often sell them as less-powerful, value-priced CPUs. In the CMP domain, one current trend is to disable under-performing cores. As the number of on-chip cores grows to 16 or more, process variation makes it more difficult to achieve uniform, high-frequency operation. Bowman and Meindl [10] find that process variation on-chip can lead to frequency reductions approximately equivalent to one step "backwards" in process technology. Heterogeneous frequencies can be designed to accommodate a specific power envelope, with high frequency cores for serial program portions, and low frequency cores for parallel portions. We examine how to schedule a single multithreaded application on a frequency heterogeneous CMP, and the benefits of running under-performing cores at lower frequencies, instead of disabling them. We leverage this work later when we try to schedule several multithreaded programs concurrently.

Intuitively, with uniform workload partitioning, one might expect low-frequency cores to reduce the performance of high-frequency cores. Additionally, one might expect computing efficiency to be better when solely running faster cores. In contrast, we find that scheduling workloads non-uniformly achieves better performance and energy consumption. Even if yields could achieve uniformly high frequencies for all cores on a CMP, we still make a case for heterogeneous processing, even if only for the embedded market (where power is a first order design concern). Our chief concern here is performance, but the energy improvements of using heterogeneous cores wisely are not insignificant. Energy is becoming a primary concern with the increased numbers

of cores and limited on-chip power budgets. Our findings may already be known to industrial practitioners, but we have yet to see such results made public.

We measure execution time to quantify improvement in delay on high-performance, multithreaded scientific codes. We allocate differing workloads, using both slow and fast cores. We assign less work to slow cores, and execute more iterations of parallelized loops on fast cores. We find:

1. Increasing thread counts and utilizing the significantly slower processors can improve overall performance.
2. Dynamic scheduling techniques can impact application performance depending on application, but are sufficient in equally dividing instructions among cores.
3. Heterogeneous chips are competitive with their homogeneous counterparts for memory-limited benchmarks in performance and energy.
4. PMCs can be used at runtime to determine if using slower cores improves performance.
5. OpenMP pragmas are better than using extra virtual threads for partitioning workloads.
6. The level of heterogeneity (frequency variation) and an application's dependence on memory are critical factors in determining whether dynamic scheduling is worthwhile.

We execute multithreaded benchmarks from NAS [3] and SPEC OMP [53] on real hardware, measuring execution time and total power consumption. Our investigation finds that using diverse cores can deliver significant speedups and reduce total energy consumption for most applications. The increased energy overhead of slower cores is offset

by the improvement in performance. Energy increases are non-linear due to energy costs of shared resources such as caches, buses, and main memory not increasing with numbers of cores. This will be important in future CMP systems, since process variations limit the feasibility of uniform cores for larger-scale CMPs.

1.4 Co-Scheduling for Frequency Heterogeneous CMPs

In addition to co-scheduling for homogeneous processors, and scheduling a multithreaded program for heterogeneous processors, there is the case of co-scheduling for heterogeneous CMPs. Processor frequencies have stopped growing at previous rates due to thermal and power issues. Transistor feature sizes however have continued to decrease, making transistor real estate cheap, leading to the development of CMPs. In order to meet a given power envelope, several processors can exist on chip, albeit at a reduced frequency and voltage compared to a single processor. Rather than reducing voltage and frequency of all cores, a fraction of cores can be targeted, leaving others at their original settings, resulting in a frequency heterogeneous CMP. A frequency heterogeneous CMP can also exist from process variation limiting maximum attainable frequency for some cores.

The advantage of a frequency heterogeneous CMP for multithreaded programs is it allows the serial or parallel portions of a program that scales badly to run on the faster processors (a subset). Remaining cores can be used for memory bound or low-priority tasks. The drawback of such a static configuration is that it is less power efficient than using more cores and an even lower voltage. However, increasing the numbers of cores increases on-chip communication, which increases power consumption without increasing performance, and not all applications scale well with increasing cores; therefore

performance and energy efficiency trends are application specific, since performance depends on multiple variables. Kumar et al. [35] find heterogeneous cores improve performance and energy efficiency in the single-threaded domain when constrained by area.

We present and evaluate scheduling algorithms for systematically creating multi-programmed workloads from a pool of multithreaded programs. Our goal is efficiently leverage frequency heterogeneous chip architectures by space and time sharing the processor among several parallel programs. Mainstream parts from major X86 processor companies currently support independent core frequencies (e.g., Intel's Nehalem and AMD's Phenom). Intel's Nehalem processor can increase the frequency of one processor at the expense of operating frequency of surrounding processors. By holistically scheduling programs that can make use of the faster and slower cores appropriately, performance and energy improvements can be achieved.

We leverage performance monitoring counters at run-time to create efficient schedules. We compare our results to running programs in isolation at maximum thread counts. We dynamically partition the workload to alleviate workload imbalances among fast and slow cores. Our scheduler is processor-aware, comparing IPCs between slow and fast processors, choosing the most efficient processor configuration for a given benchmark. Programs are paired based on their intra-core communication activity. Unlike prior work where the substrate is a result of area constraints [35], our frequency heterogenous substrate results from power and variation constraints. We use the SPEC OMP [53] benchmarks, and verify our performance and energy on an Intel eight-core shared memory system.

Our work makes three important contributions:

1. We show scheduling in a processor aware manner is more efficient than in a processor oblivious manner.
2. We show co-scheduling based on processor speeds is insufficient, and resource contention must be taken into account.
3. We find utilizing all threads is not the most energy efficient configuration when co-scheduling for a heterogeneous processor.

We tackle the problem of non-linear scaling of programs with increasing numbers of threads. We show how shared structures play a role in bottlenecking performance. We develop novel heuristics to co-schedule programs for homogeneous and frequency heterogeneous CMPs. We achieve significant improvements in performance and delay on real multicore systems.

1.5 Related Work

There has been a large body of work on the different facets of scheduling. Prior work examines scheduling for shared resources, leveraging heterogeneity, and quantitatively scaling applications and resources with memory bottlenecks. Researchers have also looked at accomodating chip variation through fixing it or scheduling around it. We briefly cover each of these areas.

1.5.1 Thread Scheduling for Multiprogrammed Parallel Applications

Prior studies have examined thread scheduling for parallel applications on large systems (composed of 32 or more processors). These are either large shared-memory systems or discrete computers that use MPI for communication among program threads. Corbalan et al. [13] examine the gains of space-sharing, time-sharing, and time and space-sharing hybrid models. This work evaluates the pre-CMP era, with little sharing of resources, resulting in no contention from co-scheduling of applications. Severance and Enbody [47] investigate hybrid time and space sharing of single and multithreaded codes. While no scheduling algorithm is determined, results indicate there are performance advantages to dynamic space scheduling single-threaded with multithreaded codes. Corbalan et al. [13] use Performance-Driven Processor Allocation (PDPA) to allocate processors based on program efficiency. Pre-determined high and low efficiency points are used to decide whether thread allocation should be increased, reduced, or remain unchanged. Decisions on expanding or reducing a program's processor allocation are taken without considering the resource consumption of other programs on the system, or the resource contention among programs. We extend this work by improving on several shortcomings. *PDPA* requires a performance analyzer that attempts to guess program run time by examining loop iteration counts and times. Also, since *PDPA* has no concept of fairness, only efficiency is maintained, and not a fair balance of core allocations. Unfortunately, since the interactions among programs are not accounted for, allocating for one program might degrade performance for neighboring applications. *PDPA* also fails to account for power, which has become a critical server design issue.

Researchers leverage the malleability of programs to change thread counts based on load to reduce context-switching and program fragmentation [14]. Corbalan et al. [13]

devise a processor driven gang scheduling scheduler that reduces context-switching by space-sharing programs that are normally time shared.

1.5.2 Thread Scheduling for Multiprogrammed Serial Applications

There is significant work on scheduling threads and applications on simultaneous multithreading (SMT) CPUs that execute multiple threads on a single processor [60]. Scheduling for SMTs is complex, since threads share the cache and all underlying resources of the processor. All threads on an SMT compete for resources, but they are not as latency sensitive as CMPs, since when one thread is bottlenecked, another thread can be swapped into the processor. This is the premise behind the architecture design for the Sun Niagara, an eight-core CMP [34]. Parekh et al. [44] predict and exploit resource requirements of individual threads to increase performance on an SMT. De Vuyst et al. [62] have looked at varying numbers of threads to reduce power consumption of less actively used cores, Snavely et al. [51] examine scheduling threads on an SMT, and conversely Shin et al. [48] look at adaptively modifying instruction fetch depending on the number of threads being executed.

For CMPs, quality of service for main memory and cache has been examined with single threaded programs running in unison [41, 42]. Several research efforts [41, 42, 55] seek to identify and combat the negative effects of parallel job scheduling of different applications together. They identify metrics for choosing single threaded programs to execute together, that have the least chance of competing for shared resources. Nesbit et al. [41, 42] implement policies that ensure fairness of resource allocation across applications, while trying to minimize performance degradation. Suh et al. [55] promote scheduling program combinations based solely on last-level cache

misses. Suh et al. [54] also propose memory-aware scheduling for multiprogrammed workloads, consisting of single-threaded programs that share the cache. Settle et al. [46] examine dynamically reconfiguring the cache for CMPs, and evaluate scheduling policies to allocate threads closer to the caches they access [57]. Fedorova et al. [20] estimate the L2 cache miss rate and then schedule suitable threads together accordingly for a CMT. This work on mitigating performance degradation runs orthogonal to our work. The scheduling, cache and DRAM channel allocation policies described can be leveraged by our scheduler to enforce resource fairness or reduce contention of co-scheduled programs.

1.5.3 Leveraging Heterogeneity

Oh and Ha [43] study static scheduling heuristics for heterogeneous processors, incorporating the effects of inter-processor communication overhead and processor heterogeneity. Kumar et al. [35] demonstrate that dynamic core assignment policies can provide significant performance gains and even outperform the best static assignment on heterogeneous multicore architectures. They find that scheduling single-threaded applications on heterogenous architectures achieves the most efficient use of area-constrained CMPs. They map a single application to one of several heterogeneous cores on a CMP, evaluating several simple heuristics (sample random core, sample neighboring core, sample all cores) for selecting the best core for each application. Since these applications are run in isolation, contention among applications is not modeled. However, this approach requires as many CMP systems as applications, since application co-scheduling is not leveraged.

De Vuyst et al. [62] examine scheduling programs across a CMP of SMTs. They

allocate appropriate resources for their workload in an effort to reduce energy while retaining performance with single-threaded programs within a multiprogrammed workload.

1.5.4 Scaling Memory-Limited Resources

Significant work has been done scaling of resources when performance counters predict the processor will be resource limited. Isci et al. [27] and Herbert and Marculescu [23] examine scaling frequency when the processor is constrained by memory bottlenecks. Conversely, work has been done on scaling the number of threads when the processor is limited [6, 16, 56]. Bhadauria and McKee [6] find the optimal thread count is not the total number of processors on a CMP, due to memory constraints. Curtis-maury et al. [16] predict optimal concurrency levels for parallel regions of multithreaded programs. Suleman et al. [56] examine optimal number of threads due to bandwidth and data-synchronization. However, none of these explore scheduling for several multithreaded programs simultaneously. Our work specifically addresses this. We do use the bandwidth aware threading method proposed, as the baseline for comparison to our work. However, this results in all applications using the maximum number of threads, as the bandwidth is never saturated.

Unlike previous work, we determine the number of threads based on efficiency for each application, accounting for the efficiency of neighboring applications. We target shared memory multithreaded programs, striving on improving performance rather than on minimizing performance degradation of individual programs that thrash with one another. We scale our resources to improve power using thread count instead of frequency, and we allocate the idle cores to running applications waiting in the batch queue. Our

work is successful due to the heterogeneity within our workloads, specifically the difference in bandwidth usage among programs.

1.5.5 Correcting Chip Variation

Practitioners compensate for variations by correcting them or by finding ways (in hardware or software) to tolerate them. Process variations cause CPUs to reach different maximum frequencies for given operating voltages, and even though Bowman and Meindl [9] find CMPs are more tolerant of variations, the problem remains important. Variations can be corrected in multiple ways [61, 58, 59] (see below), but most common is increasing operating voltage for under-performing cores, which increases power consumption. Humenay et al. [24] find that accommodating variation via voltage increases can consume 166% more power than regular numbers of cores and requires separate voltage islands per core. Donald and Martonosi [17] study techniques to turn cores on and off, adding scheduling complexity. Unlike our study, this other work examines cores running at one frequency but using differing amounts of power.

1.5.6 Accomodating Variation

Variation accommodation falls under two domains: effectively utilizing heterogeneous systems, and reducing power consumption through DVFS (Dynamic Voltage and Frequency Scaling) within CMPs or computing clusters without degrading performance. Under the first, researchers use multithreaded programs for CMPs of different processors on chip. Liu and Chaudhary [38] achieve scalable speedups with different processors working in unison by extending OpenMP and by hand optimizing codes.

Wong et al. [66] analyze load balancing program performance on a heterogeneous symmetric multi-processor (SMP) server and a cluster of SGI and Intel machines. These run parallel applications on different architectures simultaneously. They examine static scheduling via OpenMP threads and dynamic scheduling using iteration profiling. They find that a mixture of techniques improves performance but increases profiling and complexity. Balakrishnan et al. [4] examine multithreaded program performance on a prototype SMP running processors at different frequencies. They find that exposing asymmetry to the OS and programmer enables performance improvements. This SMP scalability and predictability study assumes separate chips, and thus the study does not address power implications and memory effects of CMP systems. This system has no shared caches, thus chip-to-chip latencies affect thread scaling and processor communication assumptions. Unfortunately, individual processor workloads are not revealed, making it hard to discern sources of performance bottlenecks. Both studies fail to explore the power efficiency of such systems, which we specifically target in our research.

Kadayif et al. [31] leverage heterogeneity between threads to reduce power consumption. They perform DVFS on cores independently, slowing faster threads with minimal performance degradation. Isci et al. [26] find improvements in computing efficiency to be worth the added overhead of per-core voltage islands, but the study uses a multiprogrammed workload of single threaded benchmarks in unison on a CMP. In contrast, Herbert et al. [23] use multithreaded benchmarks and find that per-core DVFS is not worth the complexity of the independent voltage islands compared to chip wide DVFS. The preferred approach appears to depend on the desired workloads. Since ours consists of multithreaded benchmarks, we target the infrastructure without per core voltage islands. Across clusters of processors, Ge et al. [22] make it clear that DVFS is worth the implementation cost, as they achieve power reductions without significant performance loss. The common thread among these studies has been the use of DVFS

to achieve energy savings while the processor is memory bound [27] to avoid significant performance degradations.

We believe ours to be the first work using real hardware to examine performance and energy for frequency heterogeneous CMPs on multithreaded, shared memory codes.

1.5.7 Heterogeneous Scheduling

Prior work examines scheduling multiprogrammed multithreaded programs with static thread counts, scheduling a multiprogrammed workload of single threaded programs on a heterogeneous CMP, and scheduling a single multithreaded program for a frequency heterogeneous CMP. We briefly cover these research areas.

McGregor et al. [39] find methods of scheduling multithreaded programs with fixed thread counts, such that the applications complement one another for CMPs composed of SMT processors. Their solution targets workloads with fixed thread counts for each program, which narrows the design space. However, by using SMT processors, programs can either be scheduled together on the same processor or on a separate core, but on the same CMP. Unfortunately, this work forces applications to remain at static thread counts, which is rarely the assumption of software programmers who design their code to work for a variety of systems that can often scale in number of cores used.

Winter and Albonesi [65] examine co-scheduling single-threaded programs on unpredictably heterogenous substrates using a combinatorial optimization algorithm to solve the assignment problem. They optimize for the lowest ED^2 of the entire system. Unfortunately, they assume no contention among programs, and optimize for the entire system, without guaranteeing fairness for individual programs. This results in some

programs having better performance at the expense of others. While they make the assumption that contention among programs is minimal, we find that this assumption does not hold for multithreaded programs, where thread communication can bottleneck application scaling. This was evident in our results in Chapter 3, which finds that contention for shared resources can significantly reduce thread throughput.

We investigate workload balancing for frequency heterogeneous CMPs in Chapter 4. Our work finds the best performance is usually achieved by using all the cores of a CMP, regardless of heterogeneity. Balancing workload chunks efficiently between slow and fast processors prevents the slow processors from bottlenecking faster ones. However, in some cases the overhead from dynamic scheduling can eclipse the performance gains, requiring run-time feedback on the balancing method chosen. We leverage these findings to create a competitive baseline for comparison, where applications use all the cores to execute their workloads, leveraging workload balancing techniques to overcome the substrate's heterogeneity.

CHAPTER 2

ROLE OF VARIABLE MEMORY LATENCY ON MULTITHREADED PROGRAM PERFORMANCE SCALING

2.1 The Multi-Dimensional DRAM Structure

Synchronous random access memory (SDRAM) is the mainstay to bridge the gap between cache and disk. DRAMs have not increased in operating frequency or bandwidth proportionately with processors. One innovation is double data rate SDRAM (DDR-SDRAM) that transfers data on both the positive and negative edges of the clock. DDR2 SDRAM further increases available memory bandwidth by doubling the internal memory frequency of DRAM chips. However, memory still operates an order of magnitude slower than core processors.

Main memory (DRAM) is partitioned into ranks, where each rank is composed of independently addressable banks, and each bank contains storage arrays addressed by row and column. When the processor needs data from memory, a request is steered to a specific rank and bank based on physical address. All banks on a rank share command and data buses, which can lead to contention when requests arrive in bursts.

Banks receive an *activate* command to charge a bank of sense amplifiers. A data row is fetched from the storage array and copied to the sense amps (a *hot row* or *open page*). Banks are sent a *read* command to access a specific column of the row. Once accesses to the row finish, a *precharge* command closes the row so another row can be opened: reading from DRAM is destructive, thus the precharge command writes data back from the hot row to DRAM storage before precharging the sense amps for the

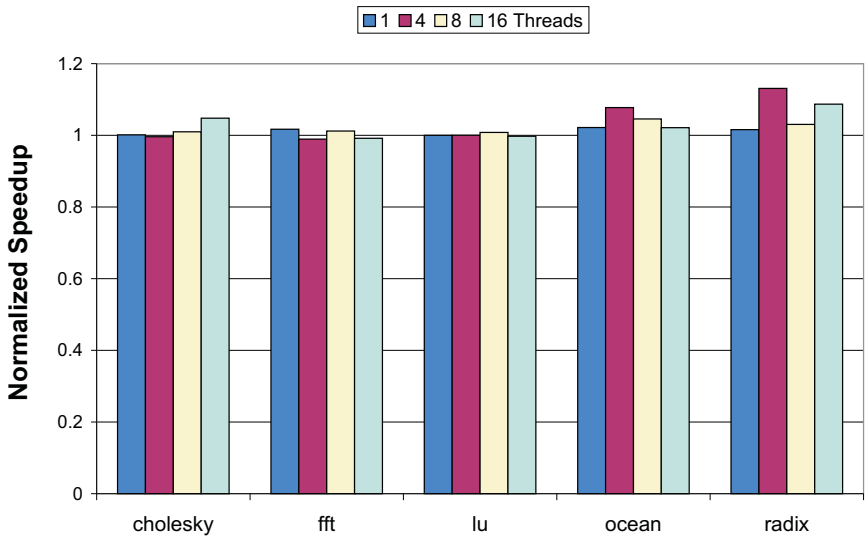


Figure 2.1: Performance for Open Page Mode with Read Priority Normalized to No Priority

next row access. This prevents pipelining accesses to different rows within a bank, but commands can be pipelined to different banks to exploit parallelism within the memory subsystem. Although banks are not pipelined, data can be interleaved across banks to reduce effective latency. For data with high spatial locality, the same row is likely to be accessed consecutively multiple times. Precharge and activate commands are wasted when the same row previously accessed is closed and subsequently reopened. Modern memory controllers usually keep rows open to leverage spatial locality, operating in "open page" mode. Memory controllers can also operate in "closed page" mode, where a row is always closed after it is accessed.

2.2 Exploiting Greater Locality

We examine performance improvements of prioritizing reads over writes in satisfying memory requests in a standard DRAM controller (now a common memory controller optimization). Figure 2.1 shows significant speedups by giving reads priority over writes for some benchmarks. Clock cycles are averaged for multithreaded configurations, and then normalized to their counterparts that give reads and writes equal priority. Differences in clock cycles among different threads are negligible. Giving reads priority generally improves performance for all but a few benchmarks. For these benchmarks, our scheduling can lead to thread starvation by giving newer reads priority over outstanding writes. Hur et al. [25] address this by having a threshold at which writes increase in priority over reads.

2.3 Experimental Setup

For this chapter of our research, we use SESC [45], a cycle-accurate MIPS ISA simulator, to evaluate our workloads. We incorporate a cycle-accurate DRAM memory controller that models transmitting memory commands synchronously with the front side bus (FSB). We assume memory chips of infinite size (limited to 4 GB in our 32-bit simulations), and no page faults. Our workloads are memory-intensive programs from the SPLASH-2 suite, which should scale to sixteen or more cores [67]. We choose programs with large memory footprints exceeding our L2 cache.

Our baseline is an aggressive CMP with four-issue out-of-order cores with relevant design parameters listed in Table 2.1. The baseline design incorporates L1 and L2 caches of 32 KB four-way and 1 MB eight-way associativity, respectively, to model

realistic silicon die area utilization, reduce program dependence on main memory, and fit large portions of the working set on chip. The fixed latency used for accessing main memory incorporates the time to perform an activate and read/write command. The precharge command time is omitted, since the memory request can be satisfied without or in tandem with the row being closed. A request takes 300 cycles, which accounts for the time to open a new row and read the data and transmit it over the FSB (Front Side Bus). When data are transmitted over the FSB, the transmission is limited by the bandwidth of the memory channel. For dynamic power consumption, we conservatively assume that cross communication between cores and clock network power is negligible. Kim et al. [32] find that based on current technology trends, the static power accounts for at least 50% of total core power, and increases linearly with number of cores and execution time. Our DRAM chips are 800MHz 4-4-4-14 memory chip timings for CAS, RAS, RAS precharge commands, with a precharge to row charge delay of 18 memory clock cycles, respectively. Our variable latency calculations for the closed and open page mode variants use discrete command timings for the row access strobe (RAS), column access strobe (CAS), and the precharge (PRE).

Our memory controller intercepts requests from L2 cache. The requests are partitioned into discrete DRAM commands. If the request's page is already open, then only a CAS command is sent to memory. Otherwise PRE and RAS commands are sent as well. Once the CAS command completes, the memory request is satisfied, and the next command is processed. Our controller implements out of order memory scheduling and read priority, which is one of the algorithms previously used by McKee [40]. Thus later requests accessing an open page are given preference over earlier requests to a different page. Read requests have preference over older write requests when choosing the next request to satisfy. We use memory buffers that are filled and satisfied on a first come first served basis (FCFS); however the FCFS priority of a request can be pre-empted if

Table 2.1: Base Architectural Parameters

Technology	70nm
Number of Cores	1/4/8/16
Execution	Out of Order
Issue/Decode/Commit Width	4
Instruction Fetch Queue Size	8
INT/FP ALU Units	2/2
Physical Registers	80
LSQ	40
Branch Mispredict Latency	2
Branch Type	Hybrid
L1 Icache	32KB 2-Way Associative 1-cycle Access 32B Lines
L1 Dcache	32KB 4-Way Associative 2-cycle Access 32B Lines
Cache Coherence	MESI
Shared L2 Cache	1024KB 8-way Associative 9-cycle Access 32B Lines
Main Memory	300 cycles static latency

Table 2.2: SPLASH-2 Large Inputs

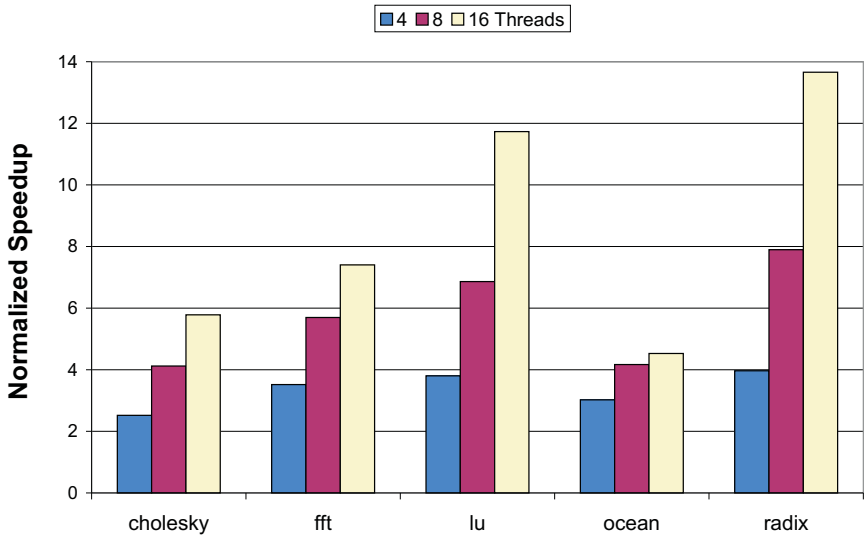
APPLICATION	INPUT PARAMETERS
Cholesky	tk29.0
FFT	20 Million
Lu	512x512 matrix, 16$\times$16 blocks
Ocean	258x258 ocean
Radix	2M keys

either of the aforementioned two conditions are satisfied. We use an infinite queue size for rescheduling accesses to remove as many configuration specific bottlenecks from our memory controller. We use a 32-bit memory space, with all addresses 32 bits long. The memory addresses are based on addresses being mapped to row address, memory channel, bank, and column address from most significant bit to least significant bit respectively. Wang [64] finds this to be the most optimal configuration, because columns should have the most variability and rows the least. Additionally, our bit mappings spread requests across the most banks, and reduce the number of row conflicts that occur for any given bank.

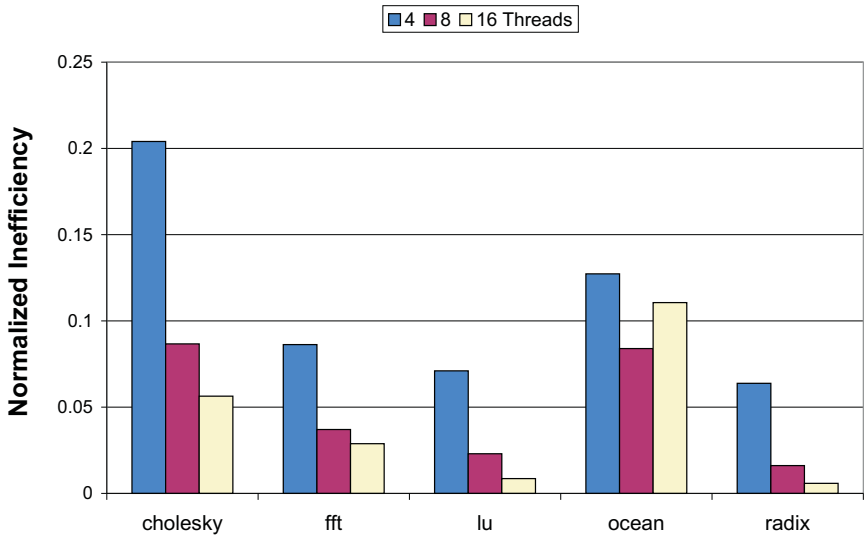
For our SPLASH-2 multithreaded benchmarks, we use a large input set. Since SPLASH-2 is rather dated, we use aggressive inputs to keep the benchmarks competitive in evaluating current performance bottlenecks on modern systems. Input parameters for our programs are outlined in Table 2.3. The larger input sets offset the initialization, program partitioning overhead, and program portions that are not parallelized.

2.4 Evaluation

For the work in this chapter, we quantitatively measure computing efficiency as the lowest energy delay squared product (ED^2) of our configurations for each benchmark. Stan and Skadron [52] find this to be a good metric for measuring power-efficient computing. Figure 2.2 (a) shows performance of programs as numbers of threads increase from one to 16. Assuming a fixed latency for every main memory request, all programs achieve a modest reduction in execution time by increasing the numbers of threads. However, even with a static latency, not all benchmarks exhibit linear improvement in performance with increasing thread counts. This is a result of: a) programs being limited



(a) Performance

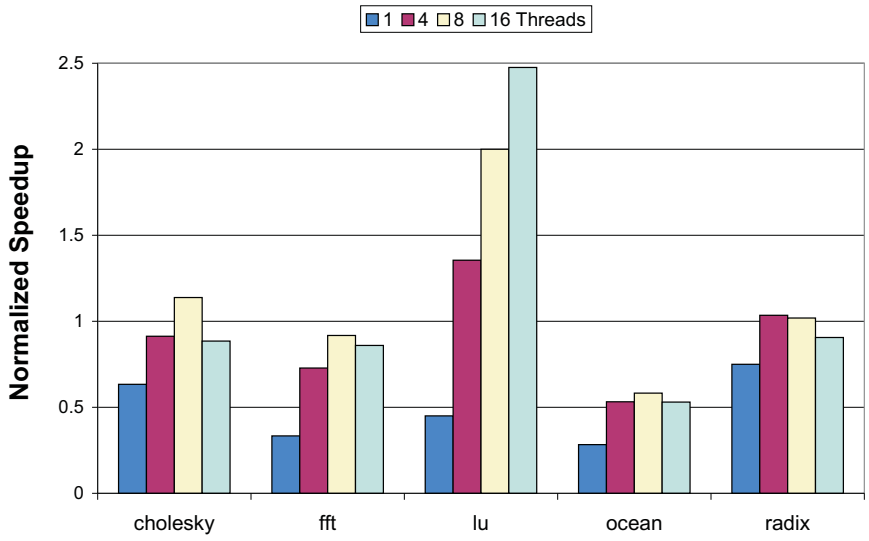


(b) Energy Delay (ED^2) Inefficiency (lower is better)

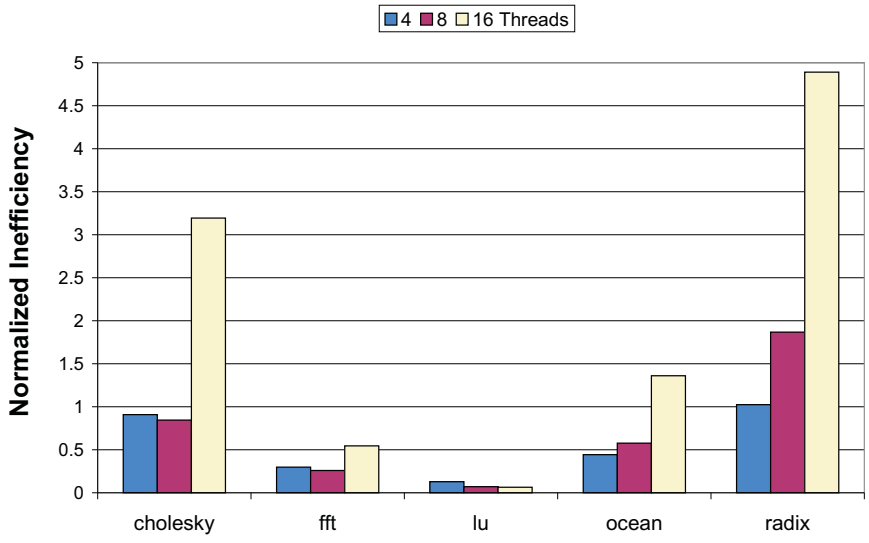
Figure 2.2: Results for Static Main Memory Latency Normalized to One Thread

by available bandwidth, and b) contention for shared data, exhibited by locks/barriers and increases in main memory accesses by contention for the shared cache. Woo et al. [67] find that for a 32 processor run, a 258 $\times$ 258 `ocean` simulation acquires and releases thousands of locks during execution. Other programs such as `radix, lu` and `fft` have no locks and few barriers, and are instead possibly limited by bandwidth. The 16-thread ocean run performs only 2% better than the eight-thread version. Bandwidth contention plays a large role here, with threads having a larger concentration of misses as thread counts increase. Figure 2.2 (b) shows the most efficient configuration for every benchmark is usually the one with the most threads. This illustrates that when assuming a static latency with peak bandwidth (every memory access being a page hit), the SPLASH benchmarks increase in speed with increases in numbers of threads and processors. These results imply that increasing numbers of threads as the solution to increasing performance efficiently, since frequency has plateaued. ED^2 results indicate increasing threads leads to improved or equivalent efficiency for most programs. `Ocean` is the exception, with the 16-thread case having a 2.6% increase in ED^2 over its eight thread counterpart, likely due to overheads of scaling and increases in cache misses. Figure 2.3 shows these results are not accurate with what a realistic system exhibits when modelled with accurate memory latencies and bandwidth.

For a single access, a closed page system and the static latency assumption both satisfy a request within the same number of clock cycles. However, once multiple accesses occur, a DRAM controller utilizing the closed page scheme quickly incurs a backlog of requests to open and close rows, often the same ones. This results in substantial performance degradations. Figure 2.3 (a) shows the performance for a system using a closed page memory system normalized to a single thread fixed latency simulation (the ideal case). Configurations ranging from one to 16 threads are graphed. Closed page mode single-threaded benchmarks perform worse than the static latency case due to the



(a) Performance

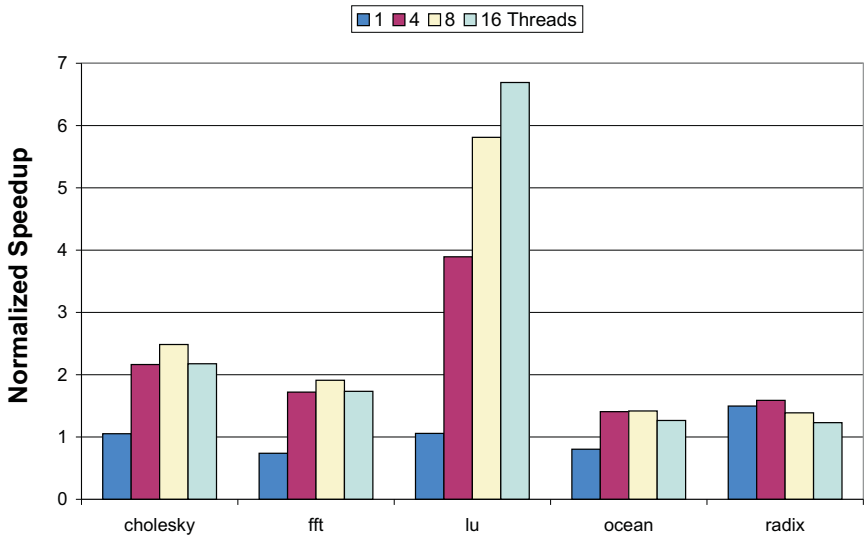


(b) Energy Delay (ED^2) Inefficiency (lower is better)

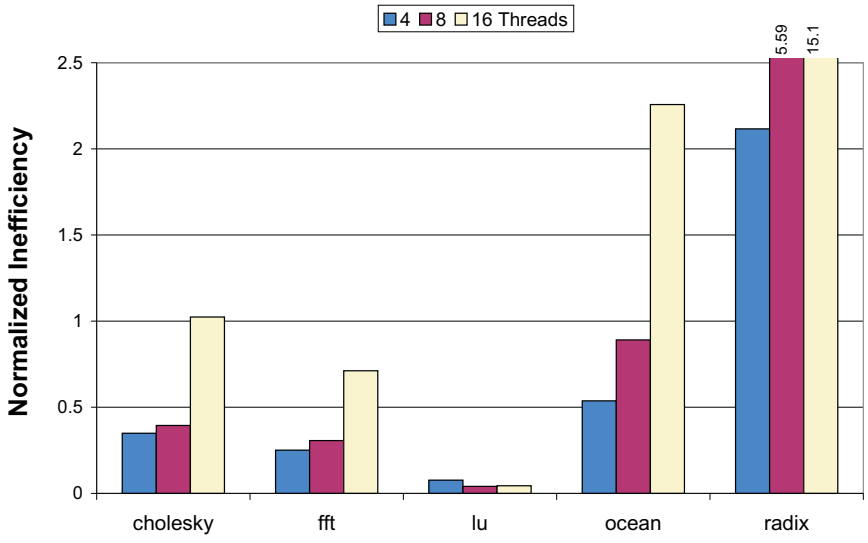
Figure 2.3: Results for Variable Closed Page Latency Normalized to One Thread

multi-dimensional structure of DRAMs. Clearly the disadvantage of this system is not only performance but power consumption as the same rows precharged are subsequently charged again. In closed page mode, almost all benchmarks are memory limited after eight or fewer threads, the exception being `lu`, which improves performance with increases in numbers of processors. Closed page simulations show a 50%+ difference (degradation) in performance compared to the static latency scenario, and performance differences are non-uniform. While all benchmarks show a degradation in performance with just one thread, the trend changes with increasing numbers of threads, depending on the benchmark. Therefore, no normalization can map results between using static latency and simulations that use a cycle-accurate DRAM timing model, because of the non-linear structure of DRAM memory. Figure 2.3 (b) illustrates the ED^2 inefficiency of closed page mode for four to 16 threads normalized to the single-threaded closed page mode configuration. It indicates that the four-threaded configuration is often the most efficient or close to it for most benchmarks. `Lu` is the only benchmark that offers strong evidence to the contrary. The differences for `cholesky` and `fft` make increasing number of threads debatable.

Figure 2.4 (a) graphs open page-mode variable latency. Performance is normalized to a single threaded configuration with static latency. Again, we see non-uniform results compared to the static latency case, with some single-threaded open page applications showing little difference in performance (`cholesky, lu`), others showing degradations (`fft, ocean`), and one showing significant improvements (`radix`). Open page mode leverages the temporal and spatial locality inherent within programs, but it fails to achieve the performance that static latency with peak bandwidth assumes would be available. This is surprising, since the aggressive out of order memory scheduling should ensure the maximum possible open page hits, but clearly it is insufficient. Examining open page performance trends, we see this issue is exacerbated with increasing threads



(a) Normalized Performance



(b) Energy Delay (ED^2) Inefficiency (lower is better)

Figure 2.4: Results for Variable Open Page Latency Normalized to One Thread

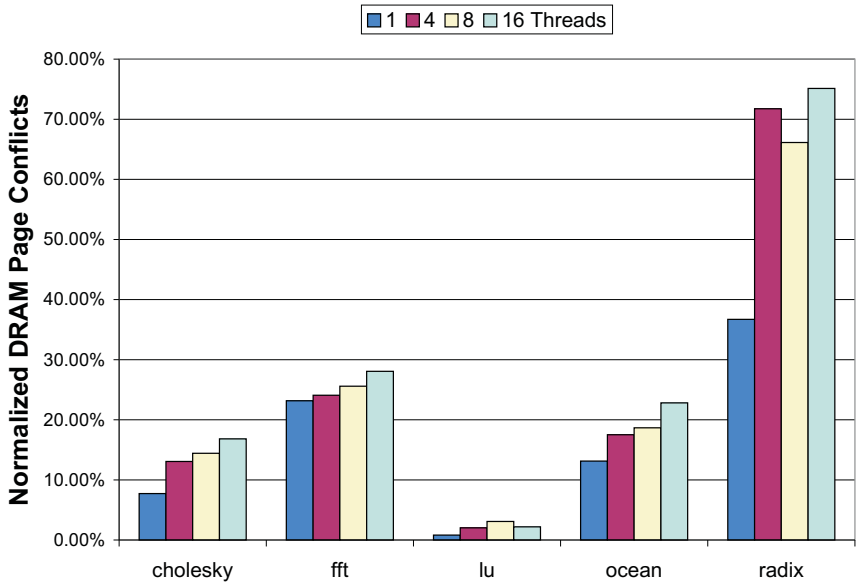


Figure 2.5: Page Conflicts Normalized to Total DRAM Accesses

as memory pressure increases on the memory channel, and more diverse and working sets compete, leading to programs bottle-necking at lower thread counts, not because of limited bandwidth, but because of latency of accessing a multi-dimensional DRAM structure.

Figure 2.4 (b) depicts the inefficiency of programs running with four to 16 threads, normalized to their single threaded counterparts (which is not shown). Surprisingly, the maximum number of threads is rarely the most efficient. For `radix`, the single threaded case is the most efficient, while for `fft, cholesky,` and `ocean`, it is the four thread configuration. If one was to always choose the maximum number of threads for each benchmark, they would end up having 339% worse ED^2 on average than the optimal value, resulting in increased energy and often delays as well. DRAM page conflicts occur when the memory requested is not on a currently opened DRAM row, requiring it to be closed, and the requested memory's corresponding row opened. Figure 2.5 shows memory page conflicts for the open page DRAM controller normalized to the

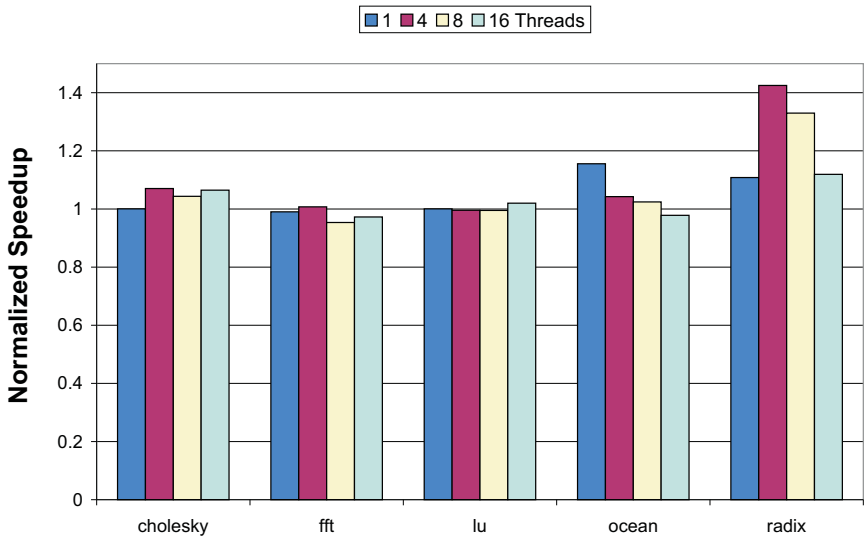


Figure 2.6: 16 Bank Performance Normalized to Eight Bank Open Page Read Priority

total number of requests to main memory. `Radix` shows a significantly high number of DRAM page conflicts, followed by `fft, ocean,` and `cholesky`, respectively. This indicates that for some benchmarks, performance degradation is due to DRAM page conflicts rather than from limited bus bandwidth. We validate these observations in Figure 2.7 where we examine benchmark sensitivity to bandwidth.

We investigate the sensitivity of benchmark performance due to number of banks and bus bandwidth speed. Although current chips consist of eight banks we are interested in sensitivity to bank size, and the expected performance improvements as this increases to 16. Increasing numbers of banks reduces contention on a single bank, since memory requests might get spread across more banks. Figure 2.6 graphs performance of 16 banks normalized to a realistic baseline memory controller that prioritizes reads, performs out-of-order memory scheduling, and utilizes an eight bank configuration. We choose such aggresive parameters to make our baseline as competitive and realistic as possible.

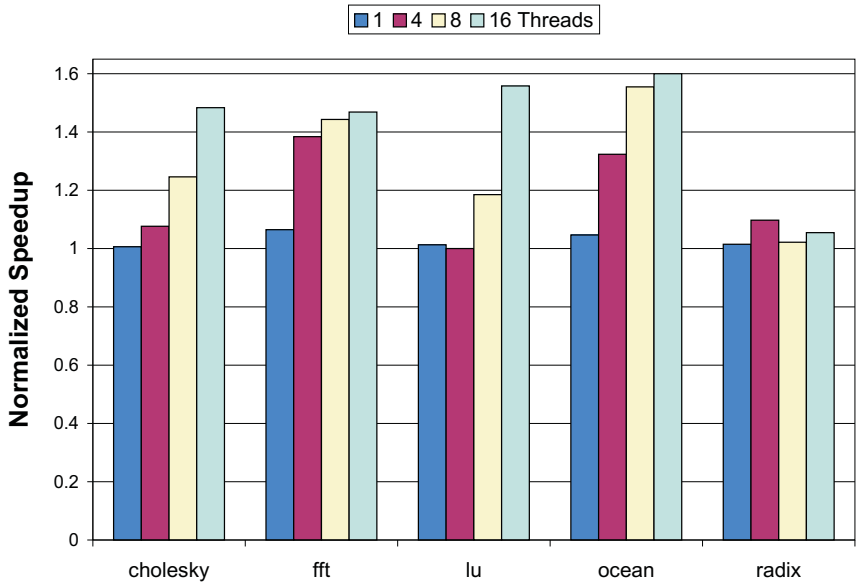


Figure 2.7: Increasing Bandwidth Performance Normalized to Open Page Read Priority

Each 16 bank configuration is normalized to its respective eight bank configuration. Performance increases from the original eight bank configuration for most benchmarks. However, some benchmarks actually degrade in performance. For example, the eight thread `fft` run increases in delay by 4.8%. This is because fewer accesses are waiting in the queue for reordering, resulting in fewer accesses to the same hot row. This results in less contention but fewer hits to a hot row where the access latency is even less than the static latency configuration. Additionally, the best performance for `cholesky` and `fft` is still the eight thread configuration, and for `radix` and `ocean` it is the four thread configuration. Therefore, doubling the number of banks fails to adequately change the thread performance curve. In fact, `ocean` saturates even faster now (at four threads rather than the eight threads with the previous eight bank configuration).

We examine performance improvements that can be expected by reducing the amount of time data occupy the bus (effectively speeding the bus). Specifically, we

assume we can double the bus speed or widen communication lanes, such that data only occupy the bus for half the number of CPU clock cycles they previously did. We graph the results of applying this optimization on our open-page memory controller, which we use as our baseline. Some benchmarks exhibit speedups (`cholesky, fft, ocean`) by increasing with numbers of threads due to increasing memory pressure. Two benchmarks show no improvements for two different reasons. `Lu` was not originally constrained by memory, so increasing bandwidth does not improve its performance. `Radix` shows negligible improvements at lower thread counts, because it suffers more from page conflicts than from limited bandwidth. This is foreshadowed in the earlier graph, which shows multithreaded radix having very high DRAM page conflict misses (over 50%). In such a scenario, it is more important to reduce page conflicts than to improve available bandwidth, since the time for satisfying a DRAM conflict is so high.

2.5 Related Work

The literature contains a wealth of research on memory devices and subsystems. We focus on DRAM latency-reduction and bandwidth-enhancing techniques here.

Researchers find that for some applications, regardless of processor operating speed, the processor is limited by the memory. Sites [50] examines this issue for single-threaded benchmarks on single core processors. McKee [40] finds one way to alleviate the speed gap between memory and processor is efficient DRAM memory scheduling. Zhang et al. [68] find DRAM does not effectively capture temporal locality of memory.

Enhanced SDRAM (ESDRAM) is a similar technology to cached DRAM, utilizing a SRAM cache row for each bank between the IO channel and the sense-amplifier [15]. The SRAM cache rows store an entire open page. The rows can be closed and another

row opened without disrupting the data being read from the SRAM row, effectively keeping the previous row open longer. Additionally, the row can be closed while the data from that row is still available for accessing, pipelining the overhead of the subsequent precharge command. ESDRAM's cache row functions like a normal open page, and can be written to and read from.

Prior research examines prefetching data to reduce the effects of the memory wall. One idea is to prefetch cache blocks when the DRAM channels are idle [37]. This method does not increase pressure on the DRAM channel since the prefetching only occurs when the channels are free. Other work dynamically modifies the TLB and cache [5] to improve memory hierarchy performance, reducing the effects of the memory wall. Another strategy is to remap non-contiguous previously unused physical addresses using directives by the application and compiler, as in the Impulse Adaptable Memory Controller [12]. This improves cache locality and bus utilization, as well as being an effective tool for prefetching from the memory controller.

The adaptive history based (AHB) memory controller reschedules memory accesses, accounting for extended dimensions of the DRAM memory hierarchy [25]. The AHB uses the same bank and row hardware conflict management schemes previously discussed. However, rank, port and channel conflicts are avoided using an adaptive history based algorithm that reschedules memory accesses based on what previous memory accesses are already in the pipeline. This is done to reduce the conflict of switching communication buses, which have significant latencies in their configuration. In the CMP domain, Nesbit et al. [41] look at running serial programs in unison on a CMP. They find that the first come first served policy that works in the serial domain is no longer optimal. They apply theoretical techniques for quality of service in networks to reduce variance in bandwidth utilization and improve system performance.

Jacob et al. [28] study DRAM at the system level and develop DRAMsim, a tool for modeling the effects of variable latency [63]. Currently, DRAMsim does not support partitioning memory requests into discrete memory commands. However, future versions by Jacob et al. [28] are scheduled to implement discrete memory controller commands, making it a viable possibility for future research. It also can currently be used by architects to examine the potential performance of their architectures when the DRAM is accurately modeled. Recently, they compared FBDIMM and DDR performance, finding good FBDIMM performance to be dependent on bus utilization and traffic [21]. Additionally, Jaleel and Jacob find [30] [29] that systems with larger reorder buffers, once modeled accurately with a memory controller, can actually degrade in performance. They find that increasing reorder buffer sizes beyond 128 entries can increase the frequency of replay traps and data cache misses, resulting in performance degradation.

We examine the effects of increasing the numbers of banks on performance. Wang found doubling the numbers of banks from one to two results in 18% performance improvement [64]; however their memory controller did not reorder memory requests. Memory reordering that achieves multiple hits to an open page results in lower latency than the static latency case. Although currently there are no high density 16 bank DDR DRAMs being shipped, Kirihata et al. [33] have shown their feasibility.

Another method of reducing the memory bottleneck is to incorporate multiple memory controllers on-chip, which is outside the scope of this study. The main disadvantages of multiple memory controllers is space limitations on chip and providing the pin count for adequate bandwidth.

2.6 Conclusions

We find that the memory wall [40] limits multithreaded program performance, and this issue is exacerbated with increasing threads. We demonstrate that memory intensive programs are limited by DRAM latencies and bandwidth as numbers of cores and program threads increase on a CMP. Even programs that show performance gains with increasing thread counts and limited bandwidth actually degrade considerably when the multi-dimensional structure of DRAM is accounted for. This bottleneck results in severely limited computing efficiency, and (depending on the application) it also increases energy or delay costs. Attempts at using state of the art memory scheduling techniques to hide DRAM latencies are not sufficient to reverse the trend. We find that using the optimal number of threads versus the maximum results in average efficiency improvements of 19.7% with memory intensive programs from the SPLASH-2 benchmarks.

We simulate closed and open page mode memory controllers and demonstrate their effects on performance. We examine performance sensitivity to DRAM parameters: specifically, increasing numbers of banks on a single channel and increasing number of bus lanes. Unfortunately neither of these achieve consistently higher thread performance. Performance can be tuned further by using a ratio of old requests to new requests and reads to writes. This work demonstrates how sharing DRAM resources among threads can lead to destructive interference and worse performance. However, such a configuration is the industry standard for multicore CMP architectures. In the next chapter, we investigate how to mitigate the effects of this shared resource contention.

CHAPTER 3

THE CASE FOR ENERGY AND PERFORMANCE AWARE CO-SCHEDULING FOR CMPS

3.1 Background

Multithreaded programs scale poorly with processor counts for many reasons, including data contention, work partitioning overheads, large serial portions, or contention for shared architectural resources. We focus on mitigating poor scaling from shared-resource contention, specifically off-chip bandwidth and memory. These bottlenecks are exacerbated by the growing gap between memory and processor speeds. However, our work can be applied to software issues such as data contention or unequal workload partitioning through run-time profiling. This poor scaling leads to inefficient hardware utilization, which can be diverted to threads from other programs for improved overall throughput. We assume the cache hierarchy described in Section 3.3, but only require that each core has a shared communication fabric at some level. We also assume that we have malleable programs (whose thread counts can change), and that each program's phases are long enough that we can sample the program and change its thread count.

Time-sliced gang-scheduling is used when executing a mixture of parallel workloads, because it improves the average response time of the jobs, processor resource requests can be completely satisfied, and it helps the working set shared by the threads [40]. It is assumed that the scheduling and context-switching overhead is negligible with respect to the interval of the time slice. Gang-scheduling all threads of a parallel program ensures that the program is not halted at synchronization points, such as barriers and critical sections, due to suspended threads. Fairness is enforced by gang-scheduling CMP resources to each program in equal time quantum. Table 3.1 shows

how time sliced gang-scheduling works for three parallel programs on a four-core CMP. Program A runs at maximum (three) thread count, gang-scheduling each of the threads on available cores in the first time quantum. After its quantum expires, program B runs, after which program C runs, and the process repeats again with program A. This continues until one program finishes, at which point it is removed from scheduling, and the remaining programs continue to swap in and out.

Table 3.1: Time Shared Gang-Scheduling of Three Parallel Applications on a Four Core CMP

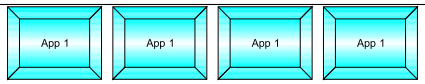
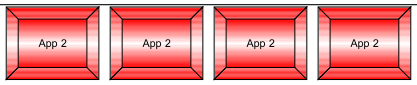
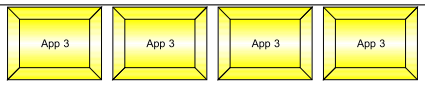
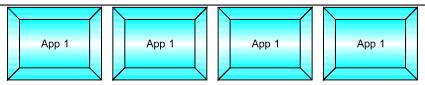
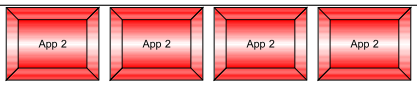
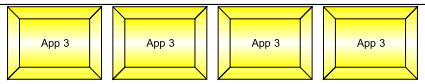
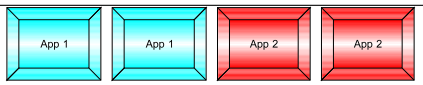
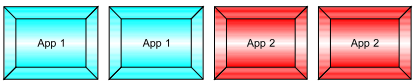
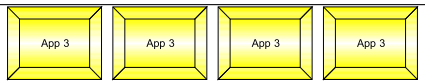
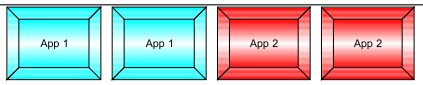
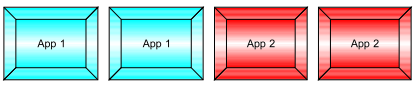
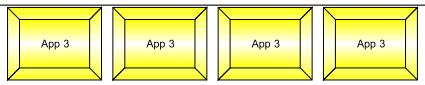
Time Quantum	Jobs Scheduled
1	
2	
3	
4	
5	
6	

Table 3.2 is one example of symbiotic time and space scheduling, with some applications being co-scheduled, and others running in isolation. In this hybrid policy, program A and program B are each allocated half the number of threads requested. This results in each program running twice as long as their normal time quanta, but on half the cores. Program C executes as normal on all cores during its time quantum. This hybrid scheduling is equally fair to all the programs since in one round-robin of time quantum (1-3), each program is given an equal amount of CPU resources.

Table 3.2: Time and Space Shared Gang-Scheduling of Three Parallel Applications on a Four Core CMP

Time Quantum	Jobs Scheduled
1	
2	
3	
4	
5	
6	

3.1.1 Conditions for Co-Scheduling

For our heuristics to efficiently co-schedule programs, we require applications to satisfy several conditions. These conditions are a lack of processors, sub-linear scaling, malleable threads, constant and long application phases and varied shared resource consumption, which are explained below.

1. There should be processor contention among applications. Processor contention occurs when there is more applications in the scheduling queue than processors available to execute all of the applications at their requested thread counts. Schedulers will time-share resources among programs, so each program receives a fast response time for resources, regardless of the execution time of the job.
2. Application queues should also consist of programs that scale sub-linearly with

increasing thread counts, since co-scheduling will not improve application scaling over gang-scheduling in isolation. If all the programs scale linearly or better with increasing threads, co-scheduling can never improve thread throughput over the baseline (as is proven in our evaluation).

3. Application threads need to be malleable, so an application can execute at thread counts fewer than those requested, without significant parallelization overhead.

4. Program phases should be long enough that any heuristic employed can converge to a solution within that time. The phase should also be long enough that the solution can be applied, and the associated overheads from converging to a solution can be mitigated with gains from co-scheduling. Therefore heuristics presented herein are not suitable for quick low throughput applications that are often idle on the desktop. Our work targets commercial applications that run for minutes and hours, and are often batch scheduled on computing clusters or render farms.

5. Applications should also fail to scale for different reasons. If all the applications within the queue are limited by the same resources at the same points in time, than co-scheduling will not alleviate this resource contention, and no benefits can be achieved.

We examine the PARSEC benchmark suite's performance and its potential for co-scheduling. PARSEC is a multithreaded suite consisting of programs designed to scale with threads, and equally distributes its workload (defined by instruction count) across threads. However, as shown in Figure 3.1 even at modest thread counts, this benchmark suite fails to scale linearly in performance. Speedup is normalized to the single-threaded case, with thread counts evaluated from one to eight. For more than half the benchmarks, performance improvements per thread degrade with increasing threads. However, benchmarks fail to scale for a variety of reasons, some are limited by memory,

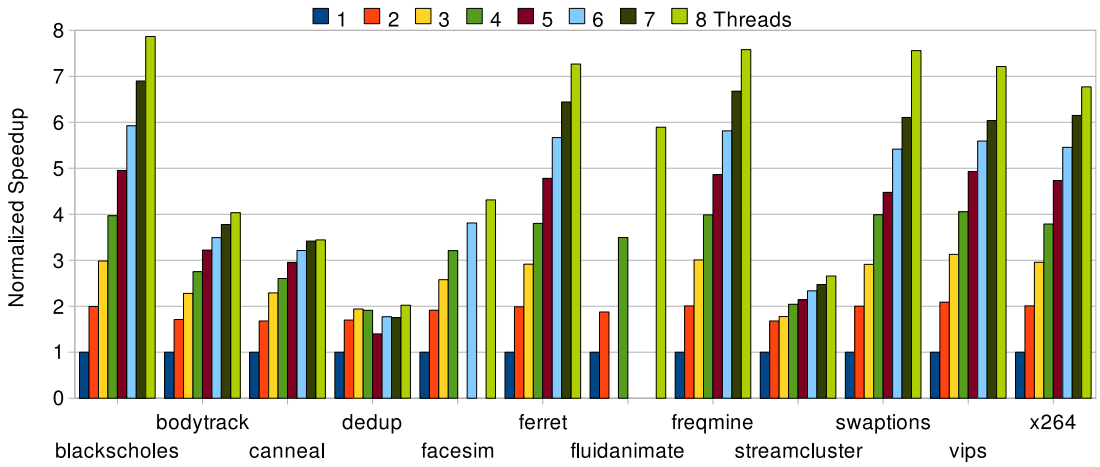


Figure 3.1: Speedup Normalized to Serial Execution (Higher is Better)

some from data synchronization, and others from contention for busses shared among private caches [7]. This indicates the potential for co-scheduling some applications to increase the efficiency of available processors. However, this requires identifying which programs are suited for concurrent execution.

3.1.2 Quantifying Shared Resource Activity

As programs increase in thread count, their working set size increases. Keeping cores fed puts pressure on the memory system. If the processor is not fed, processors stall waiting for data, which reduces thread performance. If programs share a common resource that cannot be pipelined, such as communication busses or off-chip memory, performance will suffer when they are co-scheduled. To avoid such scenarios, we quantify two different indicators of thread contention for shared resources: cache miss-rates and data bus contention. Unfortunately, lack of on-chip performance counters limits the information available at run-time on the resources a thread uses outside its own core.

Bus and cache use provides some insight, but not the complete picture, since programs can contend for resources in main memory, at the memory controller, and even the network interface. This is why our heuristic cannot guarantee that every co-scheduled set gives better performance unless such co-schedules are attempted and evaluated.

Cache miss rates have previously been proposed by Suh et al. [54] for program scheduling and cache allocation. Last-level cache miss-rates indicate what percentage of accesses result in off-chip traffic. Equation 3.1 defines the last level cache miss rate as the number of last-level cache misses normalized to total number of accesses to the last-level cache. Programs with a higher percentage of misses increase communication bus utilization and increase main memory accesses if these requests are not satisfied by other processor's caches. For our CMP, each core's L2 cache is the last-level cache, before requests are relayed to neighboring cores' L2 caches, and then to main-memory if the request is still not satisfied.

$$Last\,Level\,Cache\,Miss\,Rate = \frac{Number\,of\,Cache\,Misses}{Number\,of\,Cache\,Accesses} \tag{3.1}$$

The category of data bus contention is quantified by a vector composed of data bus occupancy and data bus wait times. Applications are ordered and numbered in relation to each other, removing the scalar difference in magnitude between the two metrics (data bus communication and data bus wait times). We choose these two metrics because all processors share the bus, and it is the last shared resource before requests go off-chip (we do not have access to PMCs on the memory controller). Data bus occupancy represents the percentage of time the data bus is busy transferring data. Equation 3.2 defines data bus occupancy as being the number of cycles data is transferred on the bus normalized to the total number of program cycles. Suleman et al. [56] use this metric to find the optimal number of threads for their bandwidth aware threading (BAT) heuristic. If

one application is commandeering the bus for transferring its data, other programs will not be able to use the bus, potentially hindering the other application's performance. We also explore data bus occupancy from main memory requests. Its definition is similar to Equation 3.2, but only data bus traffic going to and from main memory is recorded. This traffic is only from data transfers between threads and main memory, and not among threads. Applications that transfer a lot of data among cores may not hinder other applications as much as applications that utilize main memory. This is because main memory operates significantly slower than cache, and parts of it cannot be pipelined to the degree caches can, making main memory a severe impediment to performance. Depending on the performance counters supported by the hardware, further fine grained information about memory requests can be ascertained. For example, AMD quad-core processors record whether memory requests are to an open or closed page or whether page conflicts exist. However, for our benchmarks, profiling finds that most data bus traffic is due to main memory requests. Therefore, in our study, we do not discern between data bus transfers among cores or main memory.

Data bus wait time is shown in Equation 3.3 as data bus wait time normalized to total clock cycles. When the data bus is used by other processors, the requesting core must queue up to use it. As threads increase, wait times increase, as well, since there are more processors waiting to use the shared resource. Like data bus occupancy, data bus wait queues give an approximation of contention and use of a shared resource.

We profile the PARSEC suite for its average data communication, wait times and cache rates. Table 3.1.2 orders applications by increasing bus contention, and Table 3.4 orders applications by increasing cache miss rates. While, theoretically, cache and bus contention metrics might be expected to give the same ordering of applications, in practice, this is not the case. Some applications, such as `bodytrack` feature prominently

Table 3.3: Applications Ordered by Increasing Bus Contention

Application
blackscholes
swaptions
bodytrack
vips
x264
freqmine
fluidanimate
dedup
ferret
facesim
canneal
streamcluster

Table 3.4: Applications Ordered by Increasing L2 Cache Miss Rate

Application	Miss Rate
vips	1%
bodytrack	1%
blackscholes	2%
facesim	3%
swaptions	3%
x264	4%
freqmine	6%
dedup	9%
ferret	10%
streamcluster	21%
fluidanimate	38%
canneal	50%

near the top of both lists. Other applications such as `vips` are ordered differently. Data bus communication represents the ratio of cycles spent computing versus transferring data. This is not absolute, since in our super-scalar processor with prefetching, data transfers can be pipelined with independent computation. The second metric, bus wait queue sizes, depends on whether threads are trying to access the shared bus simultaneously. Miss rates are independent of the absolute time spent transferring data and independent of program execution time, therefore programs with high miss rates but relatively few memory operations are not actually bound by communication or memory. For example, `swaptions` performs relatively simple swapping operations, requiring little computation, which results in data transfers consuming a larger portion of total execution time, compared to its miss rate. `blackscholes`, a floating-point-heavy benchmark, spends a significant amount of time computing compared to its memory operations, therefore its miss rate does not noticeably affect its thread scaling. `canneal` has a very high miss rate, and most of these misses are off-chip memory accesses. While these data requests do not keep the bus very busy, off-chip accesses take longer to satisfy, resulting in its being ordered differently.

$$Data\,Bus\,Occupancy = \frac{Cycles\,Data\,Transferred\,Over\,Bus}{Total\,Cycles\,Program\,Executes} \tag{3.2}$$

$$Data\,Bus\,Queue\,Time = \frac{Cycles\,Queued\,For\,Bus}{Total\,Cycles\,Program\,Executes} \tag{3.3}$$

3.1.3 Methodology and Metrics for Hardware Aware Scheduling

We propose a framework composed of a software performance monitoring unit (PMU) and a performance monitoring user-level scheduler (PMS). The PMU examines scaling of the system during execution by sampling system performance during an application's

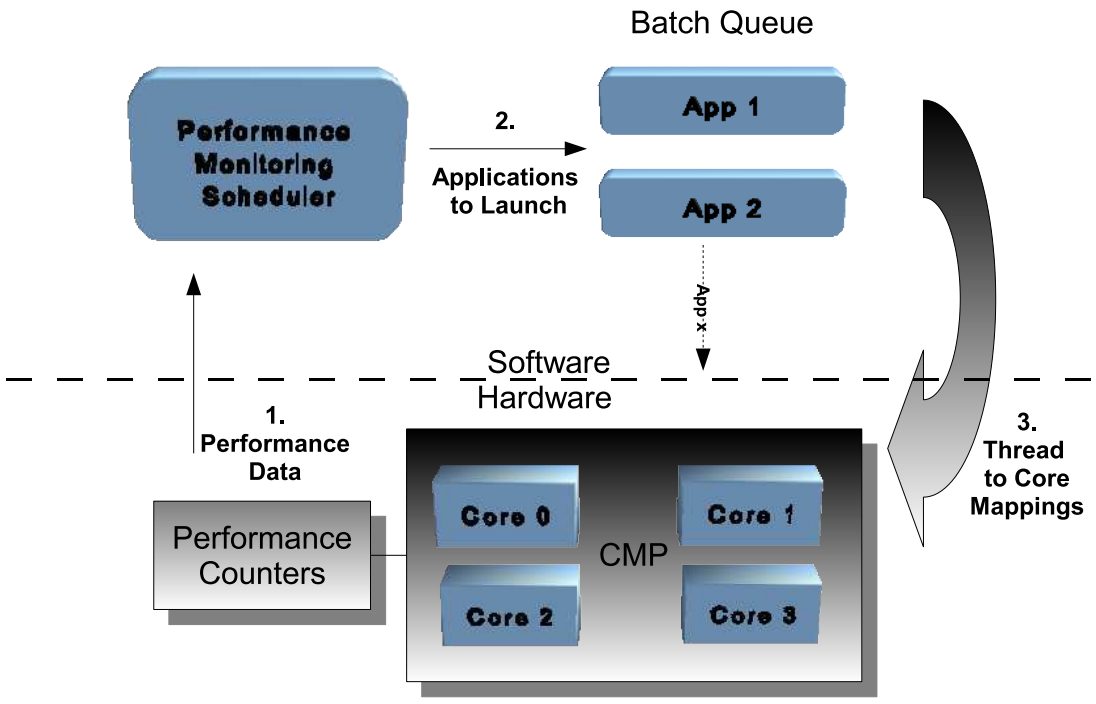


Figure 3.2: Performance Monitoring Scheduler Overview

time quantum. We feed this information to our scheduler, which reconfigures application mappings appropriately.

The PMS supervises executing programs, and dictates how many processors they can use. The scheduler incorporates feedback from interactions between software and hardware to make future scheduling decisions. Figure 3.2 shows how the hardware and software components interact. A performance monitoring unit (PMU) tracks data from performance counters. Specifically, the PMU tracks thread throughput for each program, number of threads allocated per program, and resource usage (cache miss rates and bus usage). The PMS queries data from the PMU to discern if throughput for the current application set is sufficiently high or whether rescheduling is needed. The PMU also calls the PMS when it detects an application phase change, which we discuss later.

Scheduling application threads via time and space sharing requires that the PMS know whether co-scheduling is beneficial for an application, where *beneficial* is defined as improving thread throughput. Improving thread throughput, however, is not a sufficient condition for improving application performance, since an application's increase in time allocated may not be equivalent to the space (processors) it forfeits. We define the conditions for fairness and performance improvement below.

$$Fairness = \frac{New\,Cores}{Old\,Cores} * \frac{New\,Time\,Quantum\,Size}{Old\,Time\,Quantum\,Size} \tag{3.4}$$

If a program gives up some of its cores during its quantum, it should be rewarded with extra time equal in clock cycles to the amount of space forfeited. This extra time should come from the time quantum of the application that uses the forfeited cores. This idea of fairness is quantified in Equation 3.4, where *fairness* > 1 is more than fair, and *fairness* < 1 is less than fair (unfair). New cores are the number of cores an application has after it starts sharing its space, and old cores are how many cores were originally scheduled. Old time quantum is the original size of the time quantum for each application, and new time quantum is the size once the programs are co-scheduled: time quanta are merged and co-scheduled applications are sharing the same time quantum.

$$Normalized\,Thread\,Throughput = \frac{Number\,of\,Instructions\,Retired}{Threads * ExecutionTime} \tag{3.5}$$

We define a program's thread throughput in Equation 3.5, where higher values are better. The scaling efficiency of a program is determined by number of instructions retired per second divided by number of threads. Programs that scale badly retire fewer instructions per thread as the number of threads increases when the execution time fails to proportionally decrease. Programs that scale perfectly in Figure 3.1 would show no

change in thread throughput with increasing numbers of threads. We can use this equation to compare thread counts because these programs have negligible instruction overhead with increasing number of threads (parallelizing the program does not increase the instruction count). If parallelizing programs increases instruction counts, the code can be instrumented not to count synchronization instructions [16], so instruction overheads of scaling numbers of threads do not distort the performance metrics.

Unless an application scales perfectly, it will always increase in thread throughput at a lower thread count. However, co-scheduling may increase contention, which can degrade thread throughput. We therefore quantify throughput gain in Equation 3.6 as the gain from using a lower thread count while co-scheduled with other programs, normalized to the throughput at the maximum number of threads. Both the lower thread throughput and maximum thread throughput are computed using Equation 3.5. A gain less than one means there is a loss in thread throughput at the lower thread count with co-scheduling.

$$Thread\,Throughput\,Gain\ = \frac{Throughput\,With\,Co-scheduling}{Max\,Thread\,Throughput} \tag{3.6}$$

$$Speedup = \ Fairness * (Thread\,Throughput\,Gain) \tag{3.7}$$

We define local speedup in Equation 3.7 for an application as being the product of *fairness* and thread throughput gain. If a program receives less than a fair resource allocation, but the thread throughput gain is very high, it can still achieve a speedup. If performance is greater than one, then co-scheduling achieves a performance improvement for this application.

The PMU is used to monitor PMCs and to call the PMS when a phase change occurs, since thread throughput can change within a program based on the application's

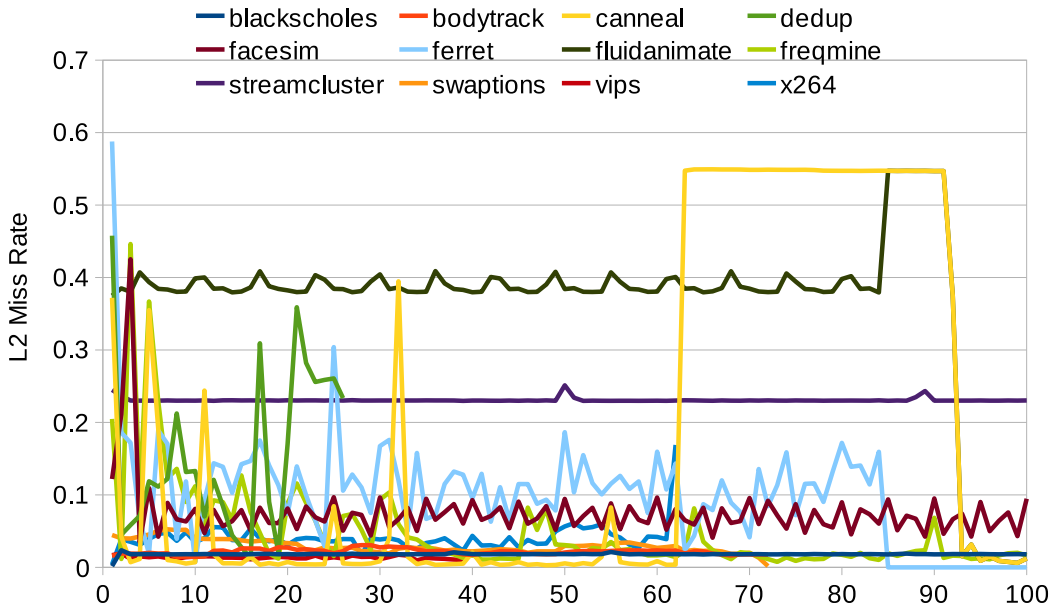


Figure 3.3: L2 Cache Miss Rate over Time

current phase. We analyze phase changes by examining events external to the core for the PARSEC benchmark suite. Figure 3.3 graphs L2 cache misses per second over program execution, where the x-axis is time in seconds. We use cache miss rates to indicate actual phase changes, since we are only interested in processor activity that uses a shared resource external to the processor. `Canneal`'s miss rates only vary between the serial and parallel portions of the program. Therefore `canneal` is considered to have only one phase for scheduling, since we are only interested in the multithreaded portion. Cache miss rates for `dedup` and `ferret` vary greatly over time. Other programs have consistent cache miss rates and have only one long program phase, when observing activity external to the processor. The scheduler can re-evaluate its decisions at new time quanta or thread spawn points, if a program's L2 cache miss rate has changed since the previous performance evaluation and the new phase remains constant (does not change for two time quanta). The new throughput values should be compared

to the optimal thread throughput level for the specific phase to determine whether co-scheduling is sub-optimal. However, this strategy requires optimal thread throughput levels for all the different phases of the program, which is not feasible. For simplicity, we compare against the throughput of using the maximum number of threads for our evaluated scheduling policies.

3.2 Scheduling

We formulate schedulers that choose programs to be run jointly, and that chooses the thread counts for each of the programs co-scheduled. Programs that are not co-scheduled are run using gang-scheduling. Our goal is to retain the thread throughput a program has at its low thread counts when it is co-scheduled with other programs. For fairness, we assume each program has equal priority, so these schedules should try to improve performance of the overall workload, while not overtly penalizing any one application within the workload (Equation $3.7 \geq 1$). We explore two different heuristics for creating schedules that space-share CMP resources. They take opposite approaches in choosing applications and application concurrency levels. The first approach strives for fairness, while optimizing for both local and global ED. It first chooses applications to co-schedule, and than application concurrency levels. The second approach optimizes for global ED throughput by co-scheduling jobs at their optimum local thread points in a resource-oblivious manner. It first chooses optimum concurrency levels for each application, and then chooses applications to co-schedule.

3.2.1 HOLISYN Scheduler

Our first approach is the HOLISYN *fair* (symbiotic) scheduler, which balances fairness of the primary job with the improved ED of running secondary jobs concurrently. It is a power-and resource-aware fair scheduler, because it restricts the minimum number of processors allocated to a program based on fairness, while optimizing for globally low ED. It improves performance and power by efficiently using resources without starving memory-intensive programs. It improves overall *throughput per watt* of the entire workload, while guaranteeing local throughput per watt of the individual programs that compose the workload.

For the scheduler to choose the appropriate programs to co-schedule, it samples each program to gather the communication ratios at the highest thread counts, and then suspends them to main memory. This phase takes place after data is loaded into main memory and tasks begin executing in parallel. This does not result in additional hardware requirements, since current schedulers already require memory and hard drive resources to swap programs when their respective time quantum elapses. Such swapping is feasible in an era of gigabytes of main memory and terabytes of hard drive space. The PMU monitors the throughput of each program at the maximum number of threads. We profile at the highest thread count to avoid cache misses that might be exhibited at lower thread counts, and to observe the resource contention at higher thread counts. After profiling, the PMS sorts applications by one of the two metrics outlined earlier(Equation 3.1 or Equation 3.2). Programs that scale almost linearly are removed from the co-scheduling queue and run in isolation since they scale well. We define a term $alpha$ to indicate how close to ideal scaling is sufficient for a program to not be co-scheduled. This requires sampling at multiple thread counts (at one and maximum thread counts), but the scheduler can work without this data, if only one data point is possible. This sampling

is not done every time quantum, but only when there is a detected phase change for more than two time quanta, so overhead is negligible. The scheduler can work without these reference data points, but it may increase number of iterations as the scheduler pairs up programs that scale well, and discards them if they degrade with co-scheduling. The scheduler initially schedules no more than two programs simultaneously: the ones with highest and lowest throughput. If there are idle cores, further (high throughput) programs can be scheduled. These scheduled programs are removed from the list of processes so they are not repeatedly scheduled, but are returned if chosen schedule fails.

There are two main components to our local search heuristic, an application chooser and a thread chooser. The application chooser picks what programs to co-schedule based on resource consumption and co-scheduling interference. The thread chooser picks what concurrency level to use for co-scheduled programs.

Initial application performance sampling creates a list sorted by resource consumption, which the application chooser uses for creating co-schedules. The list is sorted in ascending order with a pointer to the head of the list (list A) as well as a pointer to the tail of the list (for a descending order list B). The application chooser takes the following steps, and is depicted graphically in Figure 3.4:

1. It chooses the highest resource usage program (head of list B) and adds it to (an empty) co-scheduling list C.
2. It chooses the programs with the lowest resource usage (head of list A) from our set of programs, adds it to list C and removes it from list A. It divides the number of cores round-robin (sorted by lowest resource use) until no more unassigned cores remain. Therefore, if N is the total number of cores available, each process receives $N/2$ cores.
3. If the ED for all applications co-scheduled are less than what they were before co-

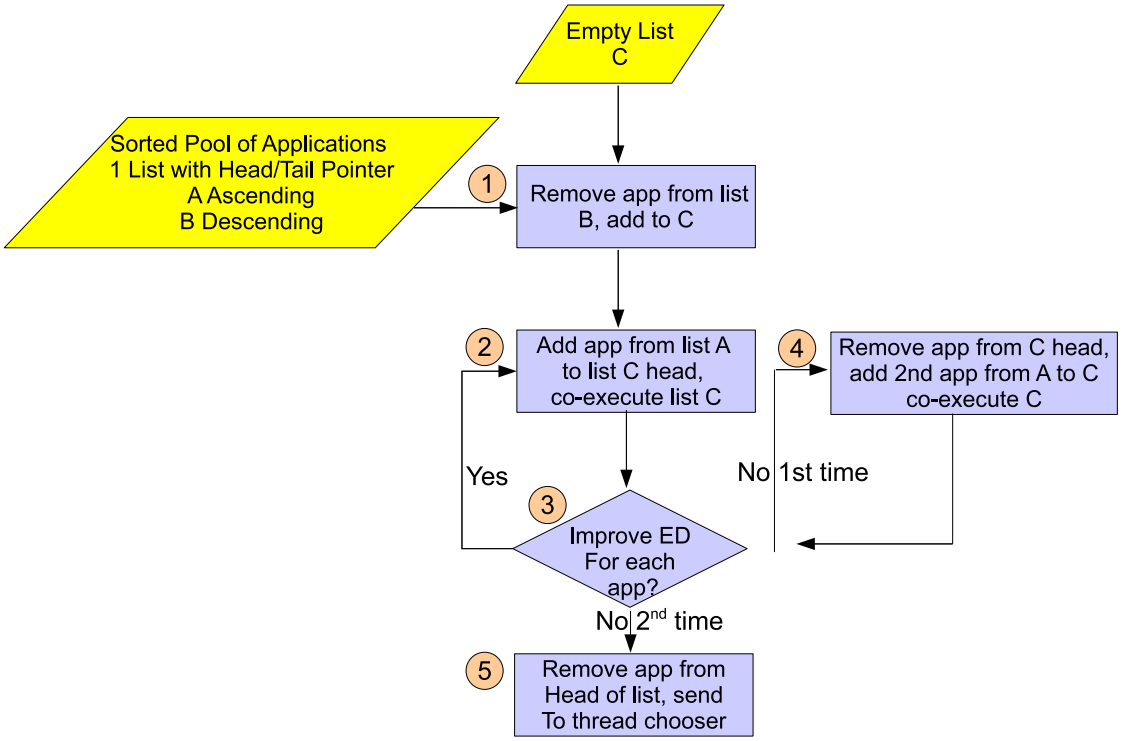


Figure 3.4: Flowchart Illustrating Application Chooser Process

scheduling, then the application most recently added to list C is removed from the head of list A, and *Step 2* is repeated. If the ED for any application is worse after co-scheduling, and this is the first failed co-scheduling attempt, than it proceeds to the next step. If this is the second time co-scheduling has failed, it proceeds to *Step 5*. It limits the number of unsuccessful application co-scheduling attempts to two, since the probability of successful scheduling decreases with later applications, due to them putting greater pressure on shared resources.

4. The program at the head of list C is removed and returned to the head of list A. The second program in list A is added to list C. It divides the number of cores round-robin (sorted by lowest resource use) until no more unassigned cores remain. *Step 3* is repeated.
5. The program at the head of list C is removed and returned to the head of list A. The

programs remaining in list C are sent to the thread chooser, and the application chooser repeats starting from *Step 1* with the set of programs remaining in lists A and C.

Attempted co-schedules consumes a full time quantum. During the exploration phases, some programs are not co-scheduled until later, due to their position in the list. These programs that are waiting for other programs (which first need to create their own co-schedules), are run in isolation at maximum thread counts until it is their turn for co-scheduling. This way no program is penalized for its ordering within the application list, or by how many applications need to be co-scheduled. We find *Step 4* of the application chooser does not change the schedules created for the benchmarks we evaluate, however we keep this secondary check since it may help for other applications, and it only marginally increases the number of time quanta needed. The application chooser steps need to occur in serial, since co-scheduled applications can change. However, the following thread chooser stages occur in parallel for different co-schedules, since the different co-schedules are independent. The thread chooser process is independent because the applications are fixed: only the respective thread numbers for each program change.

The thread chooser does the following:

1. It records the throughput of each of the programs at the equivalent thread counts from the application chooser. For two programs, it would be the throughput for equally dividing the number of cores among benchmarks.
2. It increases and decreases thread counts for each program within the list, alternating between applications. This creates two lists, a list containing the reduction in ED of each application from increasing its thread count (list A), and a list containing the increase in ED of each application from decreasing its thread count (list B).

This requires two time quanta if the list contains an even number of applications, and three time quanta if the list contains an odd number of applications. Generally, a program will scale differently, depending on which of its neighboring applications donated the thread it is using for scaling up. We make the simplifying assumption that applications scale up independently of other programs its co-scheduled with, so the thread chooser does not need to sample *C(A,A/2) (i.e., A choose* $A/2$*)* different combinations.

3. Threads are increased from applications in list A, and decreased from applications in list B. The process repeats until the applications in list B show worse ED than the baseline case of gang-scheduling, or the overall ED from selectively increasing application threads decreases. Since the process could potentially continue for N iterations with N cores, we limit the number of iterations to two to reduce number of time quanta required for a solution.

The above local-search space heuristic comes from the hill-climbing family of search space algorithms. Researchers have previously used it for finding efficient hardware configurations. Corbalan et al. [13] use hill-climbing for allocating a variable amount of processors to programs within a multiprogrammed workload. We change core counts at one core granularity, based on Corbalan et al. [13], using a granularity of four cores for a system that is four times larger.

When increasing thread counts, we must ensure that scaling up one application does not significantly penalize performance of the application being scaled down (i.e., the application whose thread is being "stolen"). In the outlined procedure for the thread chooser, each application is only checked against time-shared gang-scheduling to see if it should lose a thread. However, an application may end up losing most of its threads when co-scheduled against a program that scales almost linearly since we do not en-

force Equation 3.4. We therefore extend Equation 3.5, which is used for computing delay when comparing two time intervals of equal time periods but with different thread lengths to derive Inequality 3.8.

Inequality 3.8 specifies the condition that must be satisfied for one application to usurp processor resources, using the previously defined term $alpha$ to specify how close to ideal scaling is needed. N is the number of cores before scaling, and $N + 1$ is the number of cores after scaling up by one. Ir_t is the total CMP instructions retired before scaling over some time interval, and Ir_{t+1} is the total CMP instructions retired after scaling over an equal time interval. Inequality 3.8 serves as an additional condition when populating the list of applications for list B in *Step 2* of the thread chooser. We limit the value of $alpha$ via Inequality 3.9 to ensure that increasing the number of cores used does not appear to deliver speedups when there are none. Inequality 3.8 is a secondary constraint we add to the thread chooser, and is not required for the thread chooser stage to function.

$$\frac{Ir_{t+1}}{Ir_t * (1 - \alpha)} < \frac{N + 1}{N} \tag{3.8}$$

$$\alpha < 1 - \frac{N}{N + 1} \tag{3.9}$$

3.2.2 Greedy Scheduler

Our second scheduler uses a *greedy* bin-packing heuristic to maximize the average global system throughput in a resource-oblivious manner. It optimizes by scheduling applications at thread counts that are the most energy efficient. Applications are time-sampled at different thread counts, and the thread counts at which they exhibit the highest throughput per watt is retained. Throughput per thread (Equation 3.5) is divided by

total system power consumption, resulting in Equation 3.10. The highest throughput per watt value determines the optimal thread concurrency level for that application. These high throughput applications are then paired with additional programs, until the CMP is full (no unscheduled cores remain).

$$NormalizedThreadThroughputPerWatt = \frac{Number\,of\,Instructions\,Retired}{Threads * ExecutionTime * Watts} \tag{3.10}$$

If any cores are left idle, and no applications can fit on the left over cores, the application with the best scaling is allocated more cores so idle processors are not wasted. Once the applications are scheduled to cores, they are removed from the scheduling queues. Unlike the prior complementary algorithm, this heuristic does not try to balance fairness among programs. Programs are scheduled in a resource-oblivious manner, where their behavior is assumed to be independent of the programs they share the CMP with. The drawback of this scheduling method is that it requires extensive application profiling in advance, consisting of the number of threads and the ED at that thread level. However, most multithreaded applications lend themselves well to profiling since they consist of loops, where only a few of the loop iterations need to be run to profile the entire application [16, 56]. This allows the exploration time to be amortized over a longer workspan. The *greedy* scheduler takes the following steps:

1. It chooses the highest scaling program (highest optimum concurrency level) from our set of sampled programs, and schedules it on the CMP. If there are no remaining idle cores it is done scheduling for this program, and repeats *Step 1* for the next program. If there are remaining cores, it proceeds to the next step.
2. If the set of unscheduled applications is empty, scheduling is finished. Otherwise, the scheduler chooses the next highest scaling program from the set whose minimum processor requirement is met by the idle cores on the CMP, and schedules it

Table 3.5: Optimal Thread Concurrency Based on Lowest ED

Application	Threads
blackscholes	8
bodytrack	5
canneal	3
dedup	2
facesim	4
ferret	8
fluidanimate	8
freqmine	8
streamcluster	2
swaptions	8
vips	8
x264	8

concurrently.

3. If there are unscheduled cores remaining, *Step 2* is repeated, otherwise co-scheduling is done for this set. If no application is available where the minimum thread count is available, then thread counts of the currently scheduled programs are increased. The best scaling program's thread count is chosen to be increased until there is a lack of performance gains. It chooses the best scaling program since it has higher throughput with increasing numbers of threads, and is less likely to overtly consume shared resources.

Table 3.5 shows the thread counts with lowest theoretical ED for each application for a given phase. The applications with the lowest ED at the highest thread count

(`blackscholes`, `ferret`, `fluidanimate`, `freqmine`, `swaptions`, `vips`, `x264`) are omitted from *greedy* scheduling, since they exhibit best performance and power efficiency at the highest thread count, leaving no idle cores for scheduling other programs.

Schedulers need to be re-configured when programs change phase behavior. As mentioned in Section 3.1.3, we monitor cache misses for indications of phase changes, and find that many programs in our suite do not exhibit regular phase changes. However, if an application does exhibit changes in its shared resource consumption for two time quanta, than the respective scheduler is reloaded. For the HOLISYN class of schedulers, the resource lists are re-created, and scheduling is repeated. The *greedy* schedulers similarly repeat their samplings, and fill the CMP based on optimal individual program concurrency levels.

3.2.3 System Power Consumption

System power consumption is a function of the per core power of the CMP and the overhead of system components. Power consumption of system components such as the network adapter, main memory, main storage, and motherboard chipset (including the memory controller) are not dependent on the number of cores active. Power for each component can be divided further into dynamic and static power sources. Consolidating multiple programs on a single system amortizes the power cost of the system over several programs rather than just one. If the computing efficiency is higher when programs are run concurrently compared to when these programs are run in isolation across the CMP, then these programs finish faster. Execution time reductions lead to a reduction in static power consumption of all system components and on-chip caches. Static energy

consumption influences the efficiency point for programs differently from only looking at thread throughput, biasing it towards higher thread counts. This is because increasing thread counts usually improves performance (even if the performance improvements are sub-linear). Dynamic power consumption does not play as large a role, since when cores or system components are inactive, transistors are not switching, or are clock-gated to reduce power consumption.

3.2.4 Scalability with Many-Core Architectures

Current trends indicate future designs will contain more cores, either on chip as a CMP of multicores, or as an SMP of CMPs. We outline the scalability of the greedy and fair schedulers with increasing cores for either of the aforementioned possibilities.

The fair scheduler consists of two parts, an application chooser and a thread count chooser. The application chooser's time complexity depends solely on the number of applications scheduled, and not on the number of cores within the system. However, the shared resource metric that is used for choosing programs can change based on the configuration of the system. If the system is an SMP of CMPs, different shared resource metrics might be chosen compared to the system consisting solely of a large CMP. The thread chooser contains a hill-climbing step that varies the number of threads up to the number of cores on-chip. Increasing numbers of cores leads to linear increases in time complexity for the thread chooser. In our implementation, we cap the number of hill-climbing steps to two iterations, which works well for our system consisting of only eight-cores, however this might be too simplistic for larger systems. One solution is to increase the granularity of steps, which has been done with previous thread heuristics [13]. Corbalan et al. used a granularity of four threads for a system four times larger

than our evaluation platform for their hill-climbing algorithm. However, such a simplification keeps the time complexity at time O(n). Another method which could require fewer iterations and finer granularity is to use a binary search to reduce the search space. However, to use this method, one must assume the search space does not contain local bounds within the thread chooser step, or if such solutions exist, can be sacrificed in the interest of reducing the convergence time until a solution is reached. While outside the scope of this work, it would be interesting to see whether local minimums exist on significantly larger systems composed of many cores at the thread chooser stage. By using a divide and conquer strategy such as binary search, time complexity can be reduced to O(logN) with increasing cores.

The greedy scheduler time complexity grows linearly with increasing number of cores, since it chooses the optimal thread count at an individual application level by sampling system behavior for every single application's supported concurrency level. Unless assumptions are made about the optimal solution, the time complexity's linear dependence on N cannot be mitigated. Changing assumptions about the optimal solution, results in the chosen concurrency value no longer being guaranteed to be optimal before co-scheduling with other programs takes place. Therefore, the greedy scheduler is confined to scaling linearly with number of cores on chip.

3.3 Experimental Setup

We use the PARSEC multithreaded benchmark suite to evaluate our work. They represent a diverse set of commercial and emerging workloads. These benchmarks divide the workload evenly among threads, with theoretical linear speedups based on functional instruction traces [8]. We use publicly available tools and hardware to ensure repro-

Table 3.6: CMP Machine Configuration Parameters

Frequency	2.3 GHz
Process Technology	65 nm
Processor	Intel Xeon E5320 CMP
Number of Cores	8, dual-core SMP
L1 (Instruction) Size	32 KB 8-Way Set Associative
L1 (Data) Size	32KB 8-Way Set Associative
L2 Cache Size (Shared)	4 MB 16-Way Set Associative
Memory Controller	Off-Chip, 4 channel
Main Memory	4 GB FB-DIMM (DDR2-800)
Front Side Bus	1066 MHz

ducible results. We compile the benchmarks with the GCC 4.2 C and C++ compilers on Linux kernel 2.6.25.4 or later. We use the native input sets to mimic real scenarios. We run all benchmarks to completion on an eight core 2.3 GHz dual SMP system with four GB of FB-DIMM main memory. The front-side bus interconnect between memory and dual-core CMPs operates at 1066 MHz. Table 3.3 indicates relevant hardware parameters. We use the *pfmon* 3.2 utility from the *perfmon2* library to gather our results. The Linux *time* command accurately measures the execution times of our benchmarks to within tenths of a second. The smallest time granularity used is single seconds, since our power meter samples energy consumption per second. We use a Watts Up Pro [18] power meter to gather system power consumption. When running multiprogrammed multithreaded workloads, we use performance counter data to account for each application's per-thread power consumption. We use the Linux *taskset* command to bind our applications to specific cores, and set and change the CPU affinity for new or already running applications. We account for power consumption of individual cores via performance counter data when executing multiprogrammed workloads. Our power estimation model has been hardware-verified to estimate per core power with very low

Table 3.7: Workload Configuration with FAIRMIS Scheduling for One Phase

Workload	Threads Allocated
Bodytrack, Facesim, Canneal	3, 3, 2
Dedup, Fluidanimate, Streamcluster	2, 3, 3

Table 3.8: Workload Configuration with FAIRCOM Scheduling for One Phase

Workload	Threads Allocated
Bodytrack, Fluidanimate, Streamcluster	3, 3, 2
Canneal, Dedup, Facesim	3, 2, 3

error for single and multithreaded benchmarks [49].

We use time quanta of two seconds, to ensure each time quantum encompasses a stable period of application behavior. We loop shorter running applications until the longest running application completes (so every application completes at least once). This is due to applications having different run lengths, which can affect overall performance, since once an application completes, other programs do not need to share the CMP with it, and can complete faster. This simplification ensures the different program run times do not skew performance results. Some benchmarks require a power of two or even numbers of threads to run. In cases where there are not enough cores to satisfy a program's requirements, more virtual threads than cores are used to execute the program. If programs have many short phases, or programs do not have phasic behavior, then constant rescheduling can decrease performance benefits.

3.4 Evaluation

We apply our HOLISYN *fair* scheduler on the PARSEC benchmarks, assuming average resource usage based on the data from Tables 3.1.2 and 3.4. Ordering applica-

tions using different metrics leads to two different schedules for the application suite. We evaluate both metrics to find which works better. We define an efficiency scaling threshold which an application needs to meet to be considered sufficiently scaling such that co-scheduling is not required. Since very few programs scale perfectly, we conservatively choose scaling to be 10% of the ideal value (linear scaling with numbers of cores) at eight or more cores to be the reasonable threshold. Choosing an efficiency threshold number lower than 10% increases the scaling efficiency required for an application to not be co-scheduled. The efficiency threshold can be increased on the chance other programs can be successfully co-scheduled, but it increases the number of programs on which co-scheduling is attempted, which increases search time. `Blackscholes, ferret, freqmine, swaptions` and `vips` are omitted from concurrent scheduling, since they are within 10% of ideal performance scaling at maximum thread counts. Results on the list sorted by miss rates are denoted as FAIRMIS scheduling, and results on the list sorted by bus contention are denoted as FAIRCOM scheduling. For FAIRCOM scheduling, we briefly outline how the algorithm progresses through application scheduling of our programs sorted by bus contention for a single phase of operation. `X264` has a sufficiently high scaling efficiency that its performance does not improve with any co-scheduling. `bodytrack, fluidanimate,` and `streamcluster` are paired based on their ordering within the list and successfully co-scheduled. Similarly, `canneal, dedup` and `facesim` are co-scheduled together. This results in the workload shown in Table 3.4. Our HOLISYN *fair* scheduler based on memory misses creates the workloads listed in Table 3.4. Once all applications have been scheduled, scheduling is finished until the scheduler detects a change in resource usage (which indicates the start of the next phase).

Not all programs can be co-scheduled, although pairings are attempted with programs that exhibit low data traffic. Programs can degrade when paired with other pro-

grams because they share computing resources that suffer from destructive interference. For example, DRAM pages are a multi-dimensional structure where performance is significantly better when accesses are to an open page (as explained in section 2.1). Running multiple programs together results in page conflicts, where the current memory request lies on a different page than the one open, increasing memory latency. Programs may also have low memory requests on average, but if they saturate bandwidth at the same time as other programs, it results in increased latency, and worse performance. Our heuristic makes informed choices based on application characteristics, using real-time feedback via performance counters to ensure decisions are beneficial. However, not all contention can be measured, due to limitations of performance counter data, and our sampling granularity might not be fine enough to measure specific contention points among programs. Even with the knowledge of an individual program's resource usage, some chosen schedules may degrade performance, requiring the scheduler to adapt schedules. Unless co-scheduling is attempted for every thread count with every possible application combination, the best result cannot be guaranteed since there is no way to know when shared resources are used during a time quantum, whether they are evenly dispersed or used in unison at the same time another application needs them. This is why our schedulers needs to be able to adapt their behavior at run-time.

Table 3.9: Workloads and Threads Allocated with GREEDY Scheduling

Workload	Threads Allocated
Bodytrack, Canneal	5, 3
Dedup, Facesim, Streamcluster	2, 4, 2

Table 3.5 shows the thread counts for lowest theoretical ED for each application. The *greedy* scheduler uses this data to create the best workloads for a given phase. Applications with lowest ED at highest thread count (`blackscholes`, `ferret`, `fluidanimate`, `freqmine`, `swaptions`, `x264`, `vips`) are omitted from *greedy*

scheduling, since they exhibit best performance and power efficiency at the highest thread count, leaving no idle cores for scheduling other programs. For *greedy* scheduling, we briefly outline how the algorithm progresses through concurrent application scheduling, choosing programs to occupy all cores of the CMP. `Bodytrack` is chosen first, since it requires the largest number of threads, which leaves three cores idle in our experimental system. `Canneal` is chosen next for co-scheduling, since its optimal number of three threads matches the number of idle cores available. The three remaining applications, `dedup`, `facesim`, and `streamcluster` are all co-scheduled together on a set of eight cores, leaving no idle cores. Since all applications have been scheduled, the scheduler is done scheduling until there is a phase change in thread throughput (Equation 3.5). This results in the workload shown in Table 3.4.

We compare the performance of our scheduling with the baseline performance, which we measure by running our benchmarks at the maximum number of threads. We choose the highest thread count for several reasons: the programs are designed to use the highest thread counts available, this is the optimal number of threads based on their BAT [56] characteristics, they generally exhibit performance improvements with increasing threads, and this is the standard operating procedure in many computing environments. We compare programs based on their thread throughput efficiency. However, this also provides insight into raw performance improvements when programs need to be time-shared due to a lack of resources. For example, co-scheduling two programs, where each program originally shares the processor for equal but distinct time quanta results in the same fairness as giving each program half the cores on the CMP and merging their time quanta so they run twice as long. If the throughput is higher when a program space-shares the CMP rather than time-shares it (shown in the following graphs), then performance is improved.

Table 3.10: Workloads and Threads Allocated with PDPA Scheduling

Workload	Threads Allocated
Bodytrack, Canneal, Dedup, Facesim, Streamcluster,	2, 2, 2, 1, 1

We compare our work against Performance-Driven Processor Allocation (PDPA), which was devised and tested by Corbalan et al. [13] and found to be better than various other policies [14]. The PDPA heuristic dynamically adjusts numbers of application threads based on scaling at runtime. It uses a resource-oblivious hill-climbing algorithm to re-evaluate processor-to-thread mappings at run-time and increase or decrease the processors allocated based on the applications meeting a scaling metric. It examines application performance, making allocation decisions based on the application's scaling efficiency and fraction of processors allocated, all in isolation of other applications. Based on this algorithm, we give perfectly scaling applications their own time quantum to reduce contention, similarly to our baseline. Applications that do not scale linearly are allocated processors at runtime based on their scaling efficiency. However, it should be noted that PDPA cannot normally perform time and space sharing, and would normally co-schedule all applications concurrently. We have adapted it to time share in the case of linearly scaling programs to improve its performance. PDPA scheduling workload is shown in Table 3.10.

We show results for an *oracle* scheduler when presenting overall delay and energy for the different schedulers. The *oracle* scheduler shows the best case solution if every possible combination of space and time sharing was attempted, assuming an application is not co-scheduled more than once during a round of time quanta. This is the upper-bound on what the best performance can be expected from any possible schedulers. However, it does not investigate the case of an application being co-scheduled in multiple time quanta, i.e. application ”A” being paired with application ”B” within one

time quantum, and than application ”A” also being paired with application ”C” within a different time quantum. The *oracle* scheduler is not directly used, because the search space is large, and no polynomial-time algorithms exist for finding the best solution. Only a brute force search can guarantee the best solution (since verifying the solution takes just as long as finding the solution), which in the case of an eight-core CMP with just eight applications requires over ten thousand samples.

3.4.1 Performance

Figure 3.6- 3.8 shows thread throughput of the different co-scheduling strategies, normalized to the single-thread throughput. Higher thread throughput illustrates improvement in thread efficiency. The label *Co-schedule ideal* represents co-scheduling performance when the benchmark is run in isolation on the CMP using the same thread count as that used by the co-scheduled configuration. *Max thread* shows thread throughput when all the CMP cores are used to execute the single application. Figure 3.7 graphs thread throughput for *greedy*, where the *greedy* bin-packing algorithm is used to schedule programs at their most efficient concurrency point. Figure 3.8 graphs thread throughput for *PDPA*, where the *PDPA* scheduler is used to allocate processors to applications. Programs not co-scheduled are omitted from the graphs.

Co-schedule illustrates the throughput per thread for each application that is co-scheduled, and *co-schedule ideal* shows what the throughput is for an application if the same concurrency level is used, but the program is run in isolation. *Max thread* shows what the throughput per thread is when all the cores are used. If there is no contention, then the throughput per thread for the co-scheduled case is close to the *co-schedule ideal* value. However if there is contention, throughput decreases. The difference in

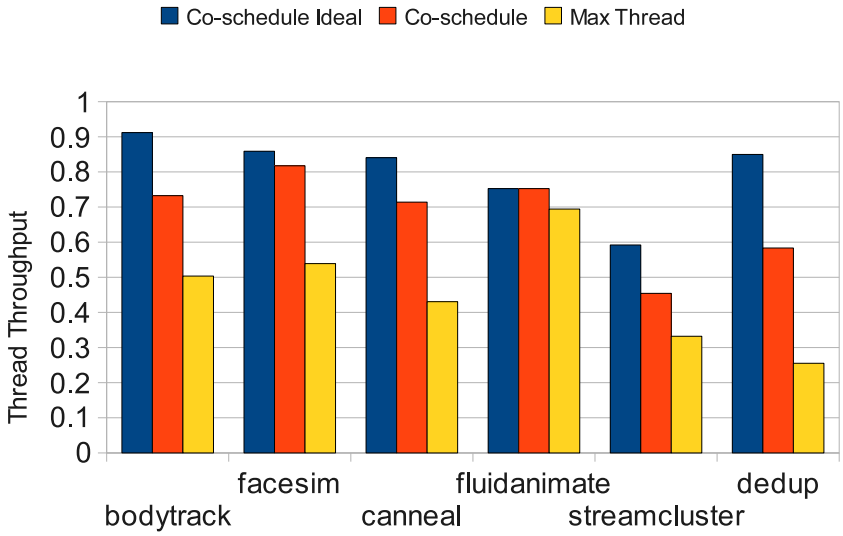


Figure 3.5: FAIRMIS Thread Throughput Normalized to Single-Thread Configuration (Higher is Better)

throughput between the single-thread and max-thread bars shows the potential speedup that can be achieved by retaining the efficiency of using fewer threads and allocating spare cores to other programs. The difference between *co-schedule* and *co-schedule ideal* shows the degradation from co-scheduling the benchmark with another program. All results are normalized to the single-threaded cases because that is the upper-bound in throughput for all cases, unless programs scale super-linearly with increasing threads (which never happens with our benchmarks).

Figure 3.5 graphs thread throughput for FAIRMIS, where the *fair* scheduler is used to divide up threads among programs and cache miss-rates are used to pair programs together. Figure 3.5 shows that `fluidanimate` and `canneal's` performance degrades minimally from being co-scheduled with other programs. This is noteworthy for `fluidanimate` since there is a significant margin for throughput to fluctuate, between the ideal and maximum thread cases. Other benchmarks degrade noticeably from their ideal values, but still remain a healthy margin above the maximum thread cases.

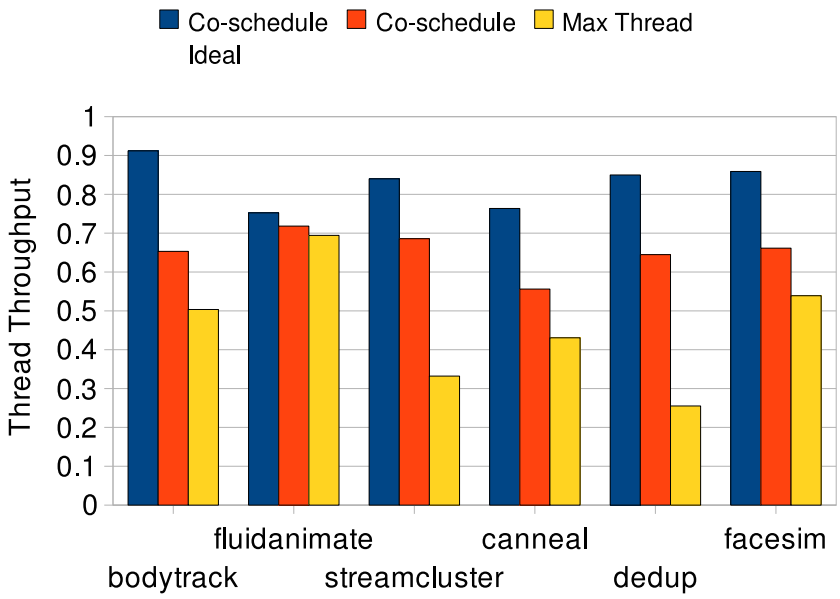


Figure 3.6: FAIRCOM Thread Throughput Normalized to Single-Thread Configuration (Higher is Better)

Figure 3.6 graphs thread throughput for FAIRCOM, where the *fair* scheduler is used to divide threads among programs paired together based on data bus usage. FAIRCOM scheduling shows results similar to FAIRMIS scheduling. While performance degrades from the ideal cases, benchmark performance is significantly better than the maximum thread cases. These improvements lead to better performance due to disproportionate time quanta. For example, in Figure 3.6, `fluidanimate` performance improves by 16% from the baseline, even though thread efficiency is only 3% better, because the increase in time quantum is higher than the loss in processors. Similarly, `streamcluster` is 2.09 times more efficient when co-scheduled rather than at maximum thread counts, but only exhibits a speedup of 1.54, because it gains less time than the processors it gives up.

Figure 3.7 shows performance with *greedy* scheduling. `Facesim` degradation from contention is worse than the maximum thread case, and had scheduling resulted in fair

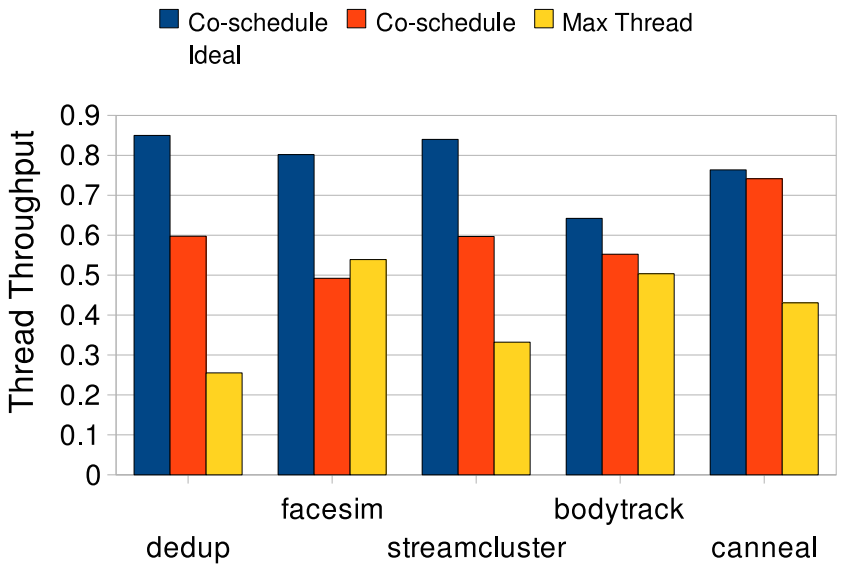


Figure 3.7: GREEDY Thread Throughput Normalized to Single-Thread Configuration (Higher is Better)

allocation of time for resources given, its performance would also be worse than the baseline. However, since from Table 3.4 we can see its time quantum increases by a factor of three, while its loss in processors is less than a factor of three, it actually speeds up in performance by 37%. The degradation from contention shows that while in isolation, an application might be the most efficient at one concurrency level, when the application is co-scheduled with others, that concurrency level may no longer be the most optimal. Other programs remain a healthy margin above the maximum thread case, keeping *greedy* competitive with other scheduling methods.

The *PDPA* performance in Figure 3.8 shows all benchmarks suffer some degradation from co-scheduling. `Bodytrack` suffers severe contention, with thread throughput dropping to the same level as the maximum thread case. `Facesim` and `streamcluster` both are only allocated a single processor. This results in `Facesim` suffering a performance loss from being co-scheduled since its efficiency over the max-

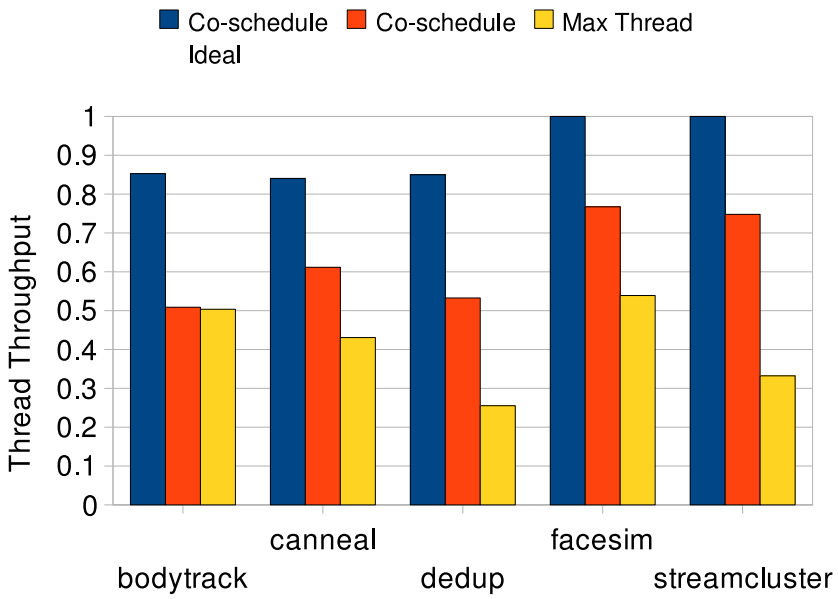


Figure 3.8: PDPA Thread Throughput Normalized to Single-Thread Configuration (Higher is Better)

imum thread case is not higher than the unfair distribution in resources it receives for sharing its time quantum. Unfortunately, the PDPA algorithm does not account for fairness or thread contention, resulting in some inefficient and unfair schedules.

The *fair* scheduler co-schedules more programs since the *greedy* scheduler is more conservative about choosing applications for co-scheduling. The *PDPA* equal partition variant tries to co-schedule as many applications together as possible, resulting in too many programs being co-scheduled, leading to the highest levels of contention.

The *fair* algorithm has several benefits. It is simpler than the *greedy* scheduler, because the application's optimal concurrency level is not required, therefore we do not need to profile the applications at every single possible concurrency level. Secondly, to use the *fair* algorithm, only the performance, cache and off-chip characteristics need to be calculated for a single (the largest) thread count. Additionally, we find that the

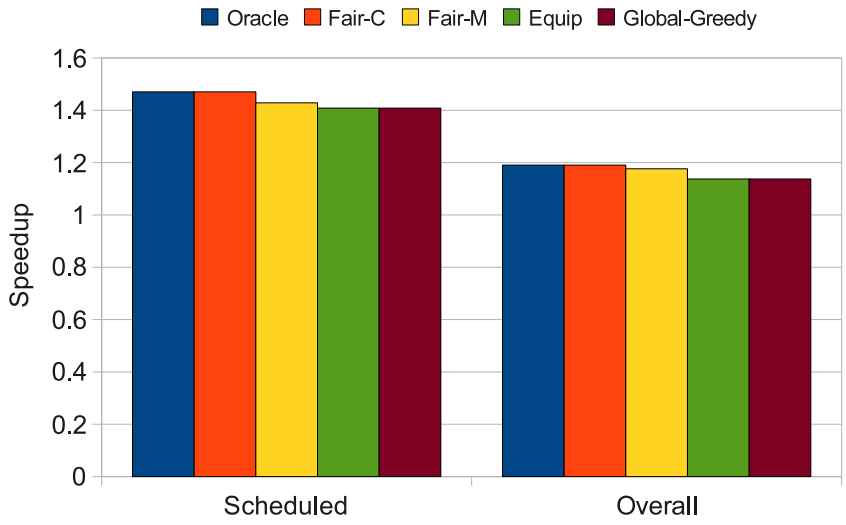


Figure 3.9: Speedup Normalized to Standalone Execution (Higher is Better)

fair algorithm can improve performance of more programs. For example, while *greedy* and *fair* achieve similar performance improvements, *fair* tries to co-schedule more programs, leading to higher overall performance.

Comparing results between FAIRCOM and FAIRMIS, we find FAIRCOM performing better for a few more benchmarks than FAIRMIS. Overall, monitoring communication latencies provides better insight into application pairing than using cache miss rates. The bus contention vector being a better indicator of sharing contention than miss rates, since the former is normalized to program running time, while memory misses are normalized to cache misses, a metric that fails to relay the application's dependence on memory for performance. For example, the three programs that scale almost linearly also exhibit the least bus contention.

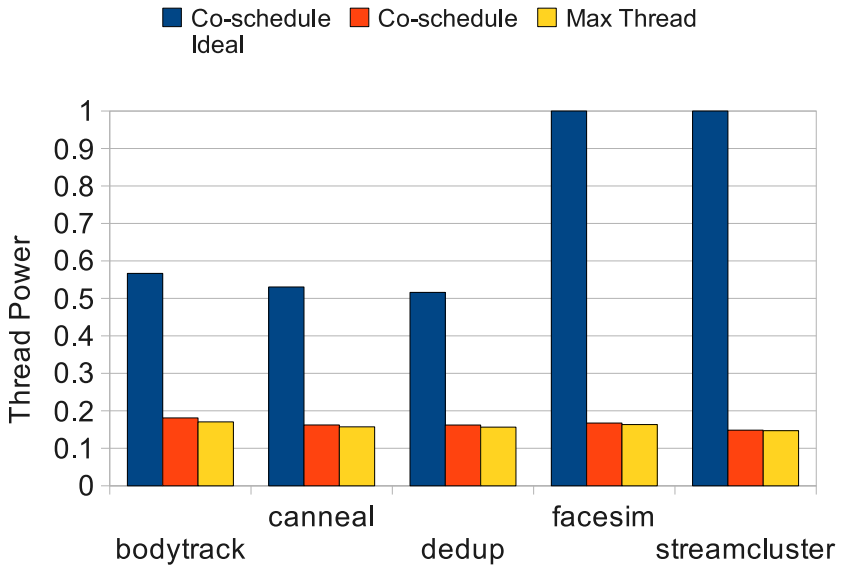
This work shows that co-scheduling helps programs retain their low thread ED efficiency on larger, multicore CMPs, by efficiently leveraging idle resources for other tasks. Figure 3.9 shows overall speedup of the different schedulers normalized to the

baseline case of time shared gang-scheduling. Label *Scheduled* shows speedup for applications chosen for co-scheduling, while *Overall* shows speedup over the entire benchmark suite, including programs not chosen to be co-scheduled. The results differ in the *Overall* case because the *Oracle* and *fair* schedulers co-schedule more programs, leading to a greater overall effect. The *Overall* results are not biased from one scheduler having a large speedup from co-scheduling only a few programs that give it the most gains and leaving out others that might bring down the average speedup achieved. The *fair* schedulers perform as well as the *Oracle* (the best case solution) when paired by bus contention, and are very competitive when co-scheduled based on memory. *Greedy* and *PDPA* show lower gains due to higher contention among scheduled programs. While the differences in speedup across co-schedulers are not significant, the performance of individual applications vary greatly since most of the schedulers do not guarantee fairness for individual applications co-scheduled.

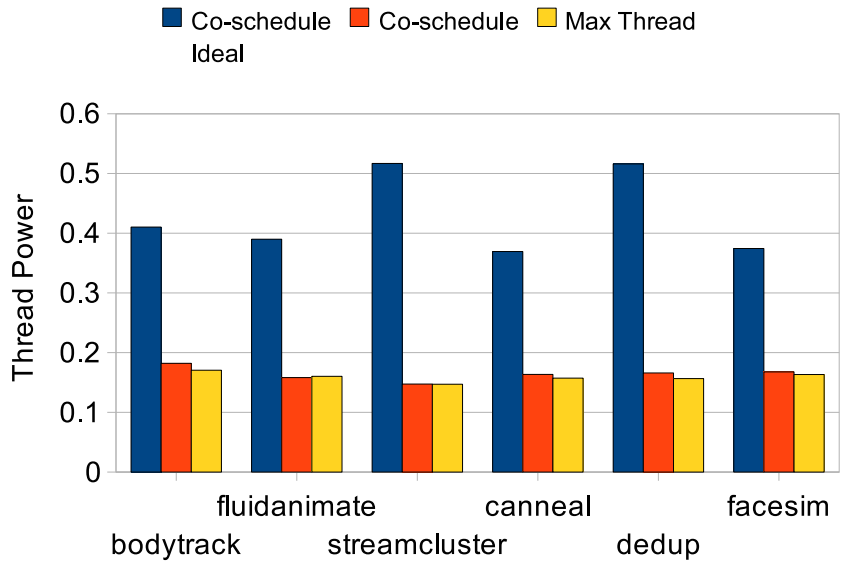
3.4.2 Power and Energy

We examine the total system power consumed by each thread of the different workload configurations. While single-threaded benchmarks have the highest throughput, it results in total system power being amortized over only a single thread. Total system power does not scale linearly with increasing numbers of active cores on the CMP, since static power of off-chip components (memory, interconnect) and on-chip memories exists while cores are idle. For example, our test system idles at 180W and reaches 300W (depending on the benchmark) when all eight cores are active. As numbers of threads increase, the static power of components is amortized over a greater number of threads.

Figure 3.10 graphs power consumption per thread for applications executed at differ-



(a) PDPA



(b) FAIRCOM

Figure 3.10: Total Power Consumption per Thread Normalized to Single-Thread Configuration (Lower is Better)

ent thread counts for (a) *PDPA* and (b) *FAIRCOM* scheduling methods. *Co-schedule* are power-per thread for benchmarks co-scheduled with other programs and *Co-schedule ideal* illustrates power per thread when benchmarks are executed in isolation but at the same thread counts as when they are co-scheduled. *Max thread* shows power per thread when executing the maximum number of threads (eight). Power per thread is normalized to a single-thread configuration where lower values are better. Power per-thread decreases with increasing numbers of threads, with the lowest power per thread cost being either the co-scheduled workloads or the workloads at maximum thread counts. Executing at lower thread counts, the benchmarks actually use almost the same or less power as when executed in isolation. This is interesting, because at lower thread counts, the programs are running at a more efficient point and committing more useful instructions, and therefore should be consuming more dynamic power. However, if the static power makes up most of the total power, then there will be minimal change in total power when the system is running at a more efficient concurrency level. Additionally, dynamic power consumption may decrease if thread contention is reduced. Unlike the maximum thread case, benchmarks are more power-efficient (performance/watt) at the lower concurrency levels, and extra cores are also used to run other benchmarks at power-efficient concurrency levels. From the graph, we can see that the watts per thread does not change from the maximum thread cases, which is interesting since more work is being performed (as was shown in earlier graphs with higher thread throughput). We omit the *FAIRMIS* and *GREEDY* graphs as they show similar trends. This shows that co-scheduling significantly reduces power per thread over the co-scheduling ideal case, since benchmarks are able to share system power consumption overhead with other programs.

We calculate energy using total power an application consumes (from our power derivation per core) and elapsed time, since energy is the product of power over time. We

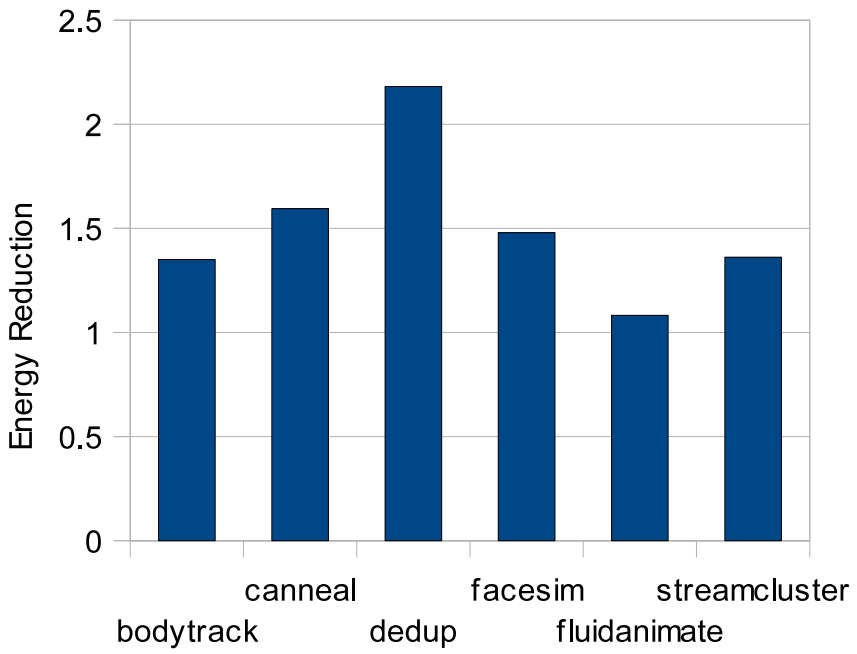


Figure 3.11: FAIRMIS Energy Reduction Normalized to Time-Shared Gang Scheduling (Higher is Better)

cannot use the delay value we calculated earlier from the improvement in performance between rounds of time quanta, since performance improvements are from programs executing for longer periods within a round of time quanta rather than shorter periods.

Figures 3.11- 3.14 graph reductions in energy (thread throughput per watt), normalized to the single-threaded case, where higher values are better. While Figure 3.10 shows that the *co-scheduled* and *max thread* runs have the lowest-power per thread, Figures 3.11- 3.14 illustrate that the co-scheduled configurations are more efficient than using the maximum threads since the throughput per thread is higher, resulting in higher power efficiency and less total energy consumption. Since the remaining cores are populated by threads from other programs that also achieve higher efficiency, power-efficiency is improved for all co-scheduled benchmarks.

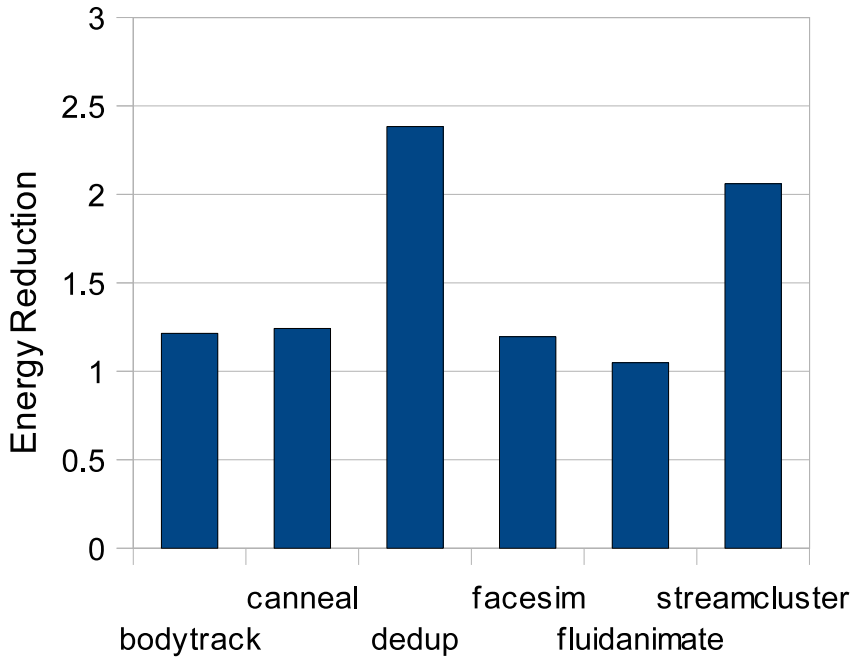


Figure 3.12: FAIRCOM Energy Reduction Normalized to Time-Shared Gang Scheduling (Higher is Better)

Power efficiency and thread throughput both directly relate to benchmark energy consumption. That is why, although not graphed the co-scheduled ideal and single-threaded benchmark runs use more energy. For example, in Figure 3.11, `bodytrack` uses 1.2 times less energy co-scheduled than when using the maximum number of threads, since benchmark throughput per thread increases, without proportional increases in power. This could have been deduced earlier from the power and performance graphs, since power per thread is the same for both cases (*co-scheduled* and *max thread*), but the throughput is much higher for the *co-scheduled* case. Throughput is even higher for the single threaded and co-scheduled ideal cases, but in those cases the power per thread (shown in Figure 3.10) is much higher. The higher power is due to the overhead of static power, and the longer running time results in substantially higher energy consumption since the application's energy is the summation of power over time. System

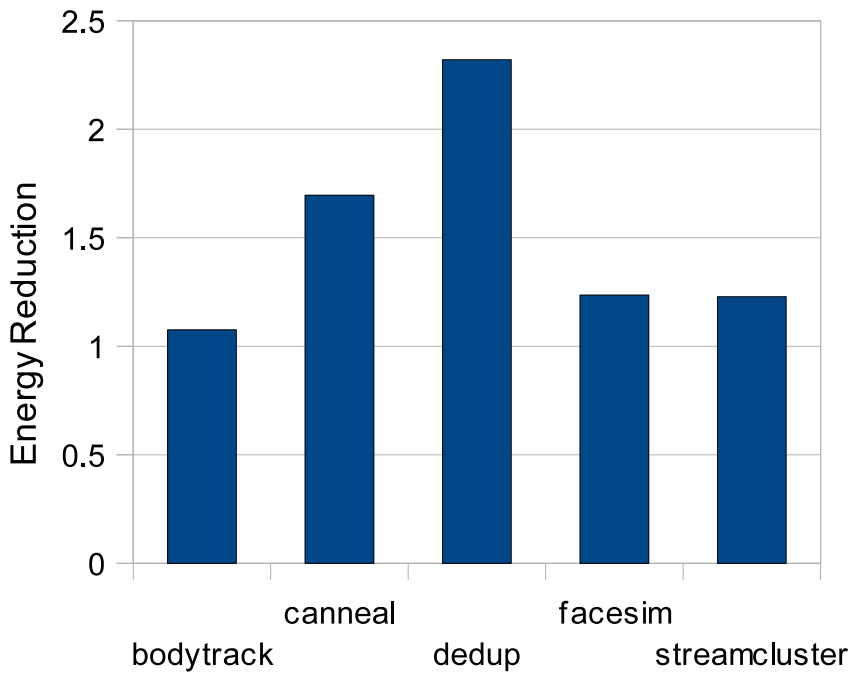


Figure 3.13: GREEDY Energy Reduction Normalized to Time-Shared Gang Scheduling (Higher is Better)

power includes not just dynamic processor power, but static processor power, memory and I/O devices. Improving thread throughput while keeping the system loaded with jobs reduces energy consumption, since the memory and chipset also consume power when the processor is memory bound and idle.

Figure 3.15 shows average energy reduction normalized to the baseline case of time-shared gang-scheduling. The label *Scheduled* represents average energy reduction for programs chosen for co-scheduling, and *Overall* shows the average reduction in energy for the entire benchmark suite. This graph is interesting, as programs that even have worse performance than the baseline could still have energy improvements (since performance degradation was from resource unfairness), and programs that improve in performance might have worse energy due to increased thrashing from contention (such as increased DRAM activity). For example, when `facesim` was co-scheduled using

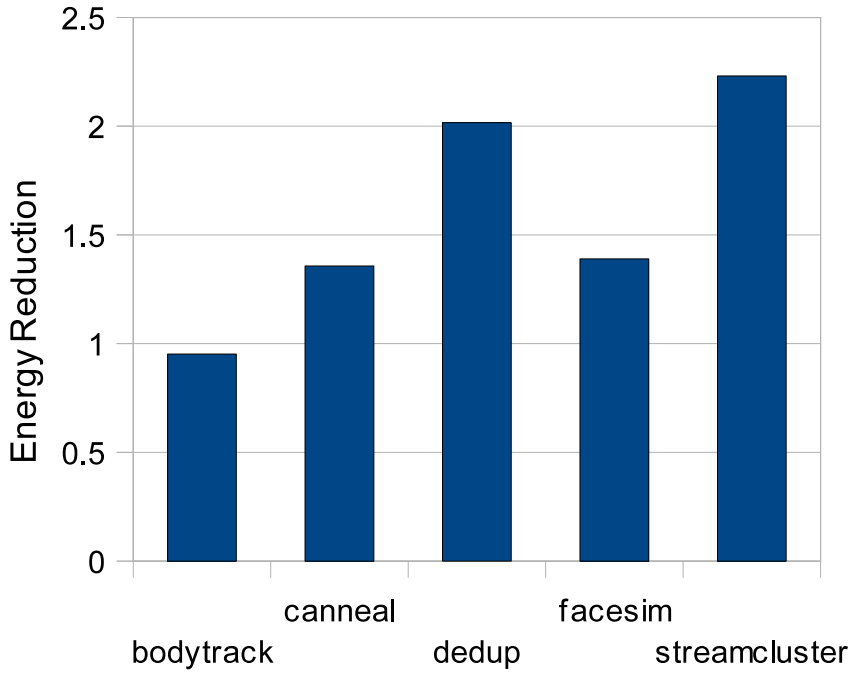


Figure 3.14: PDPA Energy Reduction Normalized to Time-Shared Gang Scheduling (Higher is Better)

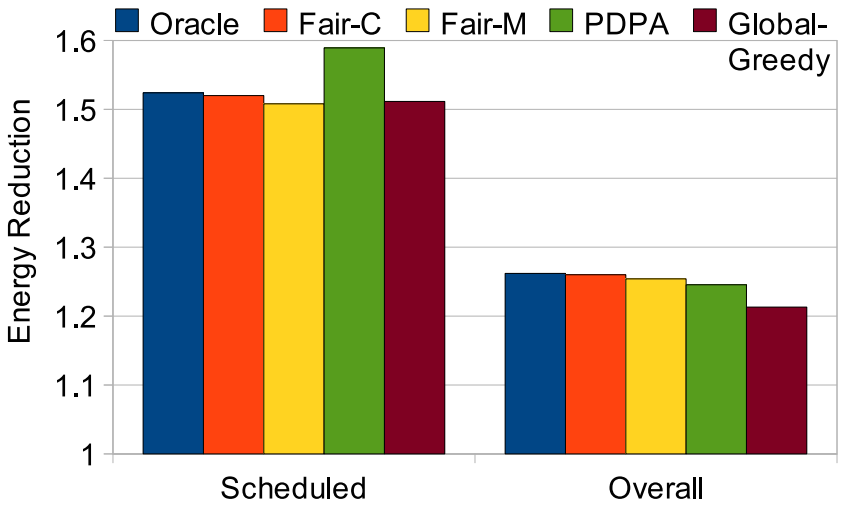


Figure 3.15: Energy Reduction Normalized to Gang-Scheduling (Higher is Better)

PDPA, performance actually degrades by 11%, but it still exhibits energy reductions of 1.39. *PDPA* scheduling shows the highest energy-reduction for programs chosen for co-scheduling, however positive reductions are achieved at the expense of some programs, with some programs such as `bodytrack` having negative improvements. *FAIR-COM* scheduling shows the second highest reduction of programs actually scheduled and highest overall, since it achieves positive reductions for all programs co-scheduled, and chooses the most programs for co-scheduling. Most noteworthy, is that overall, requiring all programs to benefit from co-scheduling does not hinder our heuristic from achieving the best performance compared to other heuristics which target optimizing for only global throughput..

3.5 Conclusions

We investigate multithreaded programs that are constrained by shared resources, and increase their collective throughput. Unlike previous work that uses frequency scaling for bandwidth limited CMPs, we reduce the numbers of threads and schedule programs together to improve individual benchmark efficiency and overall workload performance. Our scheduler improves over previous co-schedulers, minimizing contention by choosing programs that complement each other in shared resource use. Our methodology works for cases when threads fail to scale by design, as well as when they are limited by their underlying hardware. Programmers spend considerable time and effort to parallelize their programs. Unfortunately, the hardware does not always scale as desired. We profile performance and energy at run-time, which ensures our chosen schedules are adaptive and can change with program phases and behavior. We take a holistic approach to scheduling applications concurrently on an eight-core machine. Using real hardware, we devise metrics for scheduling workloads that improve over standard conventions.

We show that using the bandwidth aware threading metric (BAT) by Suleman et al. [56] does not provide the optimal thread count for performance or energy, and runtime scheduling and monitoring is required to contend with on-chip contention. By monitoring bus contention, instructions retired, and last level cache misses, we provide insight into how to schedule multithreaded applications together. We leverage these HOLISYN schedules to achieve significant performance improvements over time multiplexing the entire CMP. The advantages of our approach are that we require little knowledge of the software, and no offline access to software code, since we determine our scheduling information from hardware performance counters. We introduce two scheduling heuristics, the *fair* and *greedy* schedulers, which create multiprogrammed workloads, choosing the appropriate programs and thread counts depending on software scaling performance on the underlying hardware. Our schedulers achieve a net reduction in total energy from increasing thread throughput per watt and amortizing system overhead across multiple programs. Increasing throughput per thread also improves performance over time multiplexing the system when there are more programs than CMPs. Through co-scheduling, we reduce ED by a factor of 1.5 over not co-scheduling.

Future chapters examine scheduling for heterogenous cores within a CMP and dynamically partitioning the workloads during program execution, using runtime workload balancing techniques. The goals are to concurrently improve energy efficiency by mapping threads to the most appropriate cores and to balance fairness among programs. CMPs have entered the mainstream, and as their sizes grow, so will the importance of scheduling for them. With multiple virtual servers and cloud computing consisting of several independent environments executing simultaneously on a single chip, efficient scheduling is critical for achieving power efficiency on shared resource constrained CMPs.

CHAPTER 4

WORKLOAD BALANCING FOR CMPS WITH HETEROGENEOUS FREQUENCIES

4.1 Background

Here we discuss static and dynamic scheduling for loop-oriented, multithreaded codes and how they can be used to balance workloads on a CMP composed of heterogeneous cores.

4.1.1 Static Scheduling

OpenMP parallelizes loop bodies of Fortran or C programs marked by PARALLEL pragmas. Code in parallel regions contain independent iterations that can be computed by separate threads. Each processor is usually assigned one thread, although multiple threads may be given to cores supporting simultaneous multithreading [36] or multi-programming. By default, the OpenMP scheduler assumes that loop iterations require equal amounts of computation, and thus it statically assigns equal numbers of iterations to each thread. We refer to these as *chunks* (Equation (4.1)) (akin to blocks of iterations for scheduling parallel loops on symmetric multiprocessors [SMPs]). If some loops run longer than others during a workload *chunk*, a load imbalance occurs, and one or more threads finish before others. Since ends of parallel sections contain implicit barriers, faster threads wait at barriers, idling until the slowest thread finishes.

$$static\,workload\,chunk = \frac{total\,iterations}{number\,of\,threads} \tag{4.1}$$

```
#pragma omp parallel for
BEGIN PARALLEL FOR
    A[i]=B[i]*B[i];
END PARALLEL FOR
```

Figure 4.1: Example of a Parallelized For Loop

The first chunk is assigned to the first thread, the second chunk to the next, and so forth, until all chunks have been assigned. Each thread can amortize costs of fetching data into cache via multiple accesses to those data in subsequent loop operations and iterations. The overhead cost of partitioning the workload only occurs once.

Figure 4.1 illustrates a simple example. The code reads array `B`, computes a result, and writes that to array `A`. When a value at index `i` is fetched, other elements in the same cache line are also fetched. If the arrays are dense for a block-cyclic partitioning of iterations, memory costs of fetching elements `i+1`, `i+2`, . . ., can be amortized over multiple iterations. Alternatively, for cyclic partitioning, in which iterations are assigned round-robin to threads, false sharing occurs if threads work on adjacent indices. This is the same problem as loop scheduling for SMPs, but in a different context (we advocate block-cyclic over cyclic assignments of iterations to cores).

4.1.2 Dynamic Scheduling

For *static* scheduling, *chunk* sizes and numbers of *chunks* are pre-chosen based on numbers of threads. For *dynamic* scheduling, the user can specify chunk sizes and the number of chunks can exceed the number of threads. Chunks are assigned round-robin to available threads. New chunks are allocated when previous chunks complete. This tries to overcome the main drawback of *static* scheduling — the potential for load imbalances — by using smaller *chunk* sizes that are dynamically assigned to cores.

Guided scheduling is a variation of *dynamic* scheduling, where chunk sizes are not fixed, and the user can specify a lower-bound for sizes. Chunk sizes can be larger but never smaller than the specified lower-bound. The run-time library starts by allocating large chunk sizes, and keeps decreasing sizes until it reaches the lower limit specified by the user (or a chunk size of one iteration if no lower-bound is given), or no more unallocated *chunks* remain.

The dynamic approach minimizes false sharing and amortizes memory costs for accessing data touched multiple times within and across iterations. This approach assumes the program contains many small loop iterations and that chunk sizes will be sufficiently large to make scheduling overhead negligible, but remains sufficiently flexible to balance work between overloaded and idle cores. We examine this strategy for loop-oriented benchmarks, noting what problems arise and identifying counter measures. While *dynamic* scheduling is engineered for program loops with different computation overheads between loop iterations, here we leverage it for instances where the underlying thread hardware operates at different speeds. Latency to memory remains unchanged, but computation overhead per loop increases for slower cores. All changes are compiler-agnostic with respect to the underlying hardware, such that the workload can be appropriately allocated at run time, rather than at compile time.

Programmers typically parallelize the outer-most loop with nested loops. When programs contain many nested loops, load-balancing for just the outer-most loop may be insufficient. For example, *applu* and *lu* both contain many nested loops. Parallelizing the outer-most loop may not result in enough loop iterations to balance workloads between heterogeneous cores. When migrating to heterogeneous systems, hand optimizations (outside the scope of this paper) can help balance the system.

4.1.3 Overloading Physical Cores with Multiple Virtual Threads

One method of reducing workload size is increasing the numbers of virtual threads to be greater than the numbers of physical cores. Increasing numbers of threads reduces the workload chunk assigned to each thread, since the workload is amortized over more threads. Programs that need to scale by even numbers or by powers of two can scale using virtual threads, in order not to constrain their physical substrate by their virtual thread count limitations. However, programs with large numbers of barriers can suffer delay at synchronization points, such as waiting for all threads to catch up at barrier points or waiting for a lock to be released at a critical section. Threads in the critical path might be swapped out, resulting in CMP-wide stalling, while waiting for a thread to be swapped back to continue execution. Additionally, threads might access different areas of memory, resulting in increased DRAM page misses, and worse memory performance. While our platforms do not support simultaneous multithreading (SMT), executing multiple threads will not be inefficient due to thread swapping, since we use a shared-memory architecture, and all the threads belong to the same address space. Therefore, when another thread is swapped in, the cache and associated data need not be swapped out or invalidated.

We compare the performance of virtual threads to dynamic workload balancing, using the Linux scheduler to balance virtual threads across heterogenous physical cores. We compare performance on the four-core CMP as well as on an eight-core system.

Table 4.1: NAS Benchmark Class B Inputs

APPLICATION	INPUT PARAMETERS
bt	102x102x102, 200 iterations
cg	75000, 75 iterations
ep	2147483648 Numbers
ft	512x 256x 256, 20 iterations
lu	102x102x102, 250 iterations
mg	256x 256x 256, 20 iterations
sp	2102x102x102, 400 iterations

4.2 Experimental Setup

We compile with `-openmp` and `-static` using Intel C and Fortran compilers. For NAS v3.2 [3], we use class B large inputs outlined in Table 4.2. These fit in main memory, but are large enough to provide significant work per thread. We omit *IS*: it fails to run for larger input sets. For the SPEC OMP benchmarks [53], we use training inputs, since they provide reasonable workloads and complete in reasonable time. Native execution times vary from about one to ten minutes, depending on benchmark. We use Linux 2.6.24.2 and the perfmon2 library [19] to gather performance counter data.

Table 4.2 describes our testbed: an AMD Phenom 4-core CMP [1] with 2GB of memory and a 200GB HDD that can independently clock each core at either 2.2 GHz or 1.1 GHz within Linux. Our machine lacks separate voltage planes: all cores use the same voltage. We conservatively define a "bad core" as one that only runs at 1.1 GHz. Process variations might not always create such large discrepancies among cores, but this will identify points where a slower core clearly causes bottlenecks. Small frequency changes make it harder to tell whether faster cores are being bottlenecked due to a slower core, or whether they are constrained due to scaling factors such as paral-

Table 4.2: CMP Machine Configuration Parameters

Frequency	2.2 GHz (max), 1.1 GHz (min)
Process Technology	65nm SOI
Processor	AMD Phenom 9500 CMP
Number of Cores	4
L1 (Instruction) Size	64 KB 2-Way Set Associative
L1 (Data) Size	64KB 2-Way Set Associative
L2 Cache Size (Private)	512 KB/core 8-Way Set Associative
L3 Cache Size (Shared)	2 MB 32-Way Set Associative
Memory Controller	Integrated On-Chip
Memory Width	64-bits /channel
Memory Channels	2
Main Memory	2 GB PC2 5300(DDR2-667)

lelization overheads, thread spawning, and memory constraints. For example, a 16-core CMP with two cores operating at 20% lower frequency might limit other cores, but this might not be apparent if cores are constrained by other factors such as off-chip memory or program scaling. Our baseline case is two cores clocked at maximum frequency, and our test cases are two fast cores and one slow core (at half frequency), and two fast cores and two slow cores. For our eight-core study, we use an Intel eight-core machine, with relevant parameters outlined in Table 4.2. For the Intel system, fast cores operate at 2.66 GHz, and slow cores operate at 2.0 GHz (a 25% degradation in speed).

We use a Watts Up Pro power meter to measure total wall-outlet power consumption. The meter's measurements are updated once per second. To isolate the processor power consumption, we measure power of the idle system clocked at the lowest frequency. The processor consumes very little dynamic power in a low power state. We run our benchmarks from a networked file server to minimize hard drive activity. We disable the OS's automatic dynamic voltage and frequency scaling, and the CMP runs on a

Table 4.3: 8-Core SMP Machine Configuration

Frequency	2.66, 2.0 GHz
Process Technology	45 nm
Processor	Intel Xeon E5430 CMP
Number of Cores	8, dual-core SMP
L1 (Instruction) Size	32 KB 8-Way Set Associative
L1 (Data) Size	32KB 8-Way Set Associative
L2 Cache Size (Shared)	6 MB 16-Way Set Associative
Memory Controller	Off-Chip, 4 channel
Main Memory	8 GB FB-DIMM (DDR2-800)
Front Side Bus	1333 MHz

single voltage domain for all processors. When determining power consumption of the CMP, we subtract idle power values. Idle power includes system power (motherboard, hard drive), memory and CPU at idle, with cores clocked to their lowest frequency. We find dynamic power consumption scales linearly with increasing numbers of threads for computation intensive benchmarks (ep). This confirms idle cores' power are not being masked by the active cores. All data logging is on a secondary machine.

For the heterogeneous test case, we modify benchmarks to schedule threads at run-time. Our baseline uses pristine code without any modifications as that is most optimized for homogeneous processors with little scheduling overhead. Originally, the dynamic and guided scheduling pragmas were for cases in which loop iterations took different amounts of time. Here, we use those scheduling constructs for load balancing between slower and faster threads.

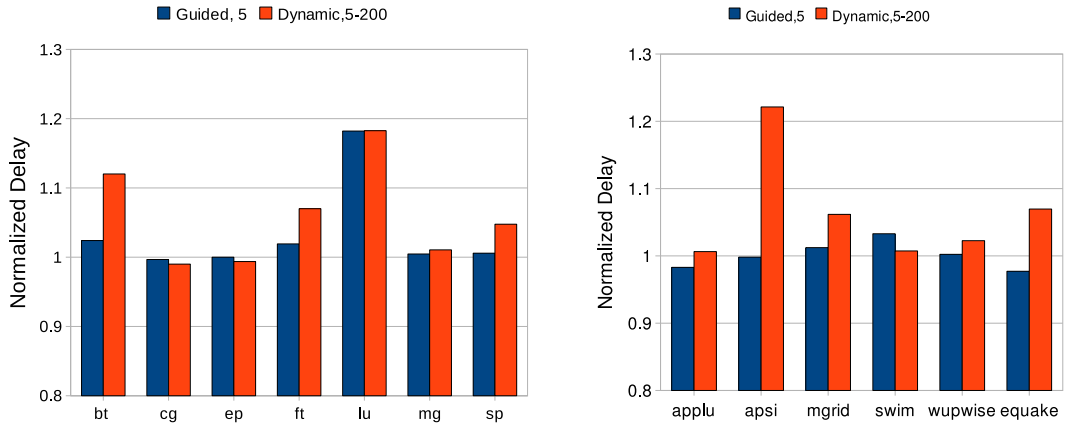
4.3 Evaluation

We examine the performance, power, energy, and run-time scheduling characteristics for homogeneous workloads and compare them against their heterogeneous counterparts at higher thread counts.

4.3.1 Performance

To gauge overhead of dynamic and guided scheduling versus static scheduling, we run `NAS` and `SPEC OpenMP` applications on three homogeneous cores and threads. Ideally, there should be no performance difference from dynamically load-balancing workload chunks among cores. Figure 4.2 shows delay increases from switching to dynamic or guided scheduling, with results normalized to our static baseline scheduler. *Guided,5* represents using guided scheduling with a minimum chunk size of five. *Dynamic,5-200* represents running all benchmarks using dynamic scheduling with fixed chunk sizes of 5, 10, 50, 100, 150 and 200 loop iterations, reporting the best performance from this range.

With NAS and SPEC benchmarks, changing dynamic chunk sizes significantly affects performance, depending on the benchmark. There is almost no change in performance with the guided scheduler, except for `lu`, which exhibits an 18% increase in delay from dynamic scheduling versus static scheduling. This reduces the benefits of dynamically allocating work for the third thread of this benchmark. Overall, we find guided scheduling to perform better than dynamic scheduling with chunk sizes, since the guided scheduler uses larger chunk sizes when possible. Guided scheduling only reduces the chunk size when backlogs occur. We find the dynamic scheduling performs

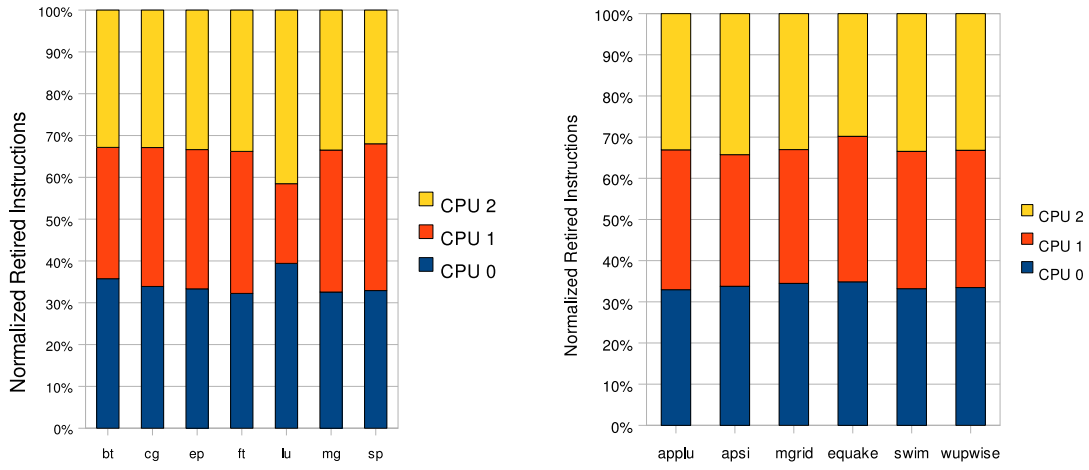


(a) NAS (b) SPEC-OMP

Figure 4.2: Scheduling Performance Normalized to Default Static Scheduling (Lower is Better)

worse for several benchmarks for which guided scheduling with just one lower bound value works well (for instance, `bt`, `ft`, `sp` and `apsi`, even when multiple chunk sizes are attempted for different programs with dynamic scheduling). Small chunk sizes work well for codes that have large nested loops, while loops without nests benefit most from larger chunk sizes, as there is less memory contention among threads.

To gain insight into these performance differences, we examine how instructions are distributed across processors using the guided scheduling policy. Ideally, each core should equally receive a third of the total instructions. Figure 4.3 graphs instruction distribution across the homogeneous processors, where CPU 0, CPU 1, CPU 2 are the three cores on the CMP. All the SPEC-OMP benchmarks show excellent scaling, with instructions being distributed equally among cores (about a third of the total instructions for each core), and only a 1% variation. The NAS benchmarks show more variation for two benchmarks. With `bt`, CPU 0 performs 5% more work than CPU 1 and 2% more work than CPU 3. `Lu` shows even higher variation, with CPU 0, CPU 1 and CPU 2 executing 39%, 19% and 42% of the total instructions respectively. This load imbalance



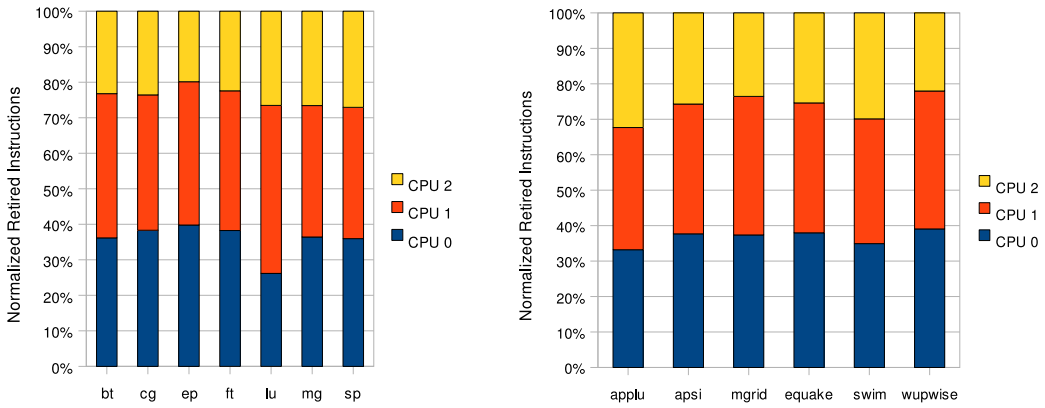
(a) NAS (b) SPEC-OMP

Figure 4.3: Distribution of Instructions Across Homogenous Cores

leads to the performance degradation graphed in Figure 4.2(a), since static scheduling (our baseline) does not suffer from this phenomenon.

With perfectly scaling benchmarks that are computation bound (rather than memory bound), increasing from two to three threads should lead to a performance increase of 50%. However, since the third thread is only half as fast as the other two threads, only a 25% performance increase is expected. Since programs rarely scale perfectly and are not always computationally bound, this 25% improvement in performance is an expected upper limit of our results. Even the 25% upper bound may not be reached if the benchmarks are not partitioned equally at run-time among the cores.

We examine the distribution of instructions across heterogeneous cores. Figure 4.4 graphs instructions across heterogeneous cores, where CPU 0 and CPU 1 operate at maximum frequency and CPU 2 runs at half the maximum frequency (1.1 GHz). Ideally, the fast cores should execute 40% of total instructions each, and the third core should process 20% due to the different operating frequencies between processors. Un-



(a) NAS (b) SPEC-OMP

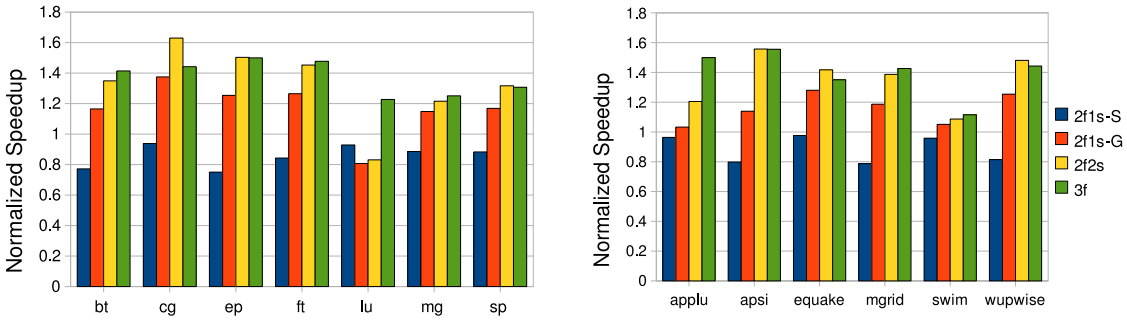
Figure 4.4: Distribution of Instructions Across Heterogenous Cores

fortunately, this perfect distribution only occurs for `ep`. On average, the slower core actually processes 25% of total instructions. This is 5% more total chip-wide instructions than the ideal case, and 25% more for that one core. This imbalance reduces the efficiency of the heterogeneous configuration. The load imbalance is worst with `swim` and `applu` where CPU 2 executes 30% and 32% of the total instructions, respectively. This does not degrade performance for `swim` significantly since it is already memory bound at the higher thread count. `Swim` has the largest memory footprint of all the SPEC benchmarks and traditionally does not scale well with threads going from two to four threads [2]. `Lu` displays a different form of imbalance between the two homogeneous cores. For most benchmarks the homogeneous cores distribute the remaining workload equally, but with `lu`, CPU 1 executes 47%, and CPU 0 and CPU 2 execute 26% and 27% of the total instructions respectively.

Speedups of running benchmarks on the heterogeneous three-core system are graphed in Figure 4.5(a) for NAS and Figure 4.5(b) for SPEC OMP. Results illustrate four different cases normalized to the two processor high frequency architecture baseline. *2f1s-S* is the heterogeneous CMP (composed of two fast cores and one slow core)

with static scheduling for thread workloads. *2f1s-G* is the guided scheduling of thread workloads on a heterogeneous three-core CMP, and *2f2s-G* is the guided scheduling of thread workloads on a heterogeneous four-core CMP (composed of two fast cores and two slow cores). *3f* is the three-thread homogeneous case where all processors run at the maximum frequency. Results are normalized to their two-thread counterparts. *3f* and *2f2s* perform similarly, with *3f* doing better for benchmarks with workload imbalances (`applu, lu`) and *2f2s* doing better for benchmarks whose cache footprint decreases with increasing threads (`cg`). *2f1s-G* performance is between *2f1s-S* and *3f*. For memory constrained benchmarks, *2f1s-G* performs competitively with *3f*. For computation limited codes, *2f1s-S* performs poorly, since the slower core bottlenecks the faster ones, resulting in an effective raw clock speed of 3.3GHz compared to the baseline's 4.4 GHz (25% slower). Performance degradation varies with benchmark, depending on thread barriers and synchronization points. *2f1s-S* is not included in further discussion, since its results are obvious and offer little insight into desirable power and performance. The NAS benchmarks with the *2f1s* setup benefit significantly from the increase in threads. Although the extra processor runs at half the frequency of the other cores, effective workload balancing ensures it does not throttle performance at synchronization points. The *2f2s* configuration also can perform as well as the *3f* configuration, effectively compensating for the one fewer fast cores with two slow ones.

The exception to our positive performance gains is `lu`. Its performance degrades on heterogenous CMP configurations, by 19% (for *2f1s-G*) and 17% (for *2f2s*). There are several reasons for this. One reason is that faster cores have to wait for the slower core at barrier points, leading to the faster cores being idle. Another reason is the synchronization overhead of dynamic chunk sizing, which can slow the system significantly. Figure 4.2 confirms our previous observations of reduced performance from dynamic scheduling overheads. Figure 4.4 confirms the workload imbalance between cores.



(a) NAS (b) SPEC-OMP

Figure 4.5: Performance Normalized to Two Thread CMP (Higher is Better)

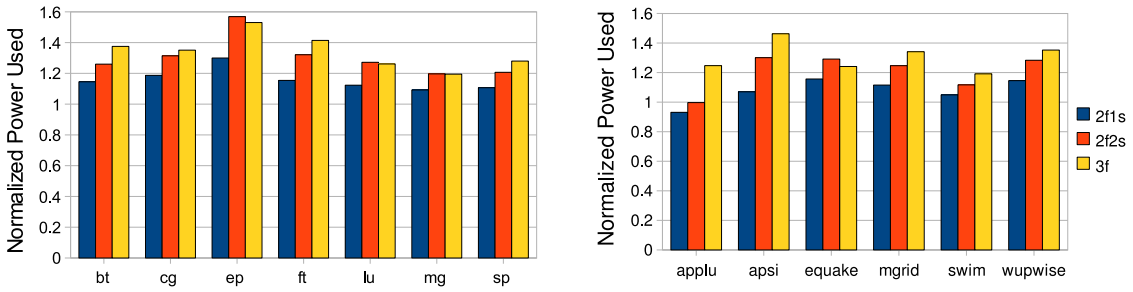
2f1s-G's `cg` benchmark shows the most gains of the NAS benchmarks, with a 38% super-linear gain in performance. `Cg`'s performance gains are due to decreasing cache misses with increasing threads. From the SPEC suite, *2f1s*'s `equake` shows the most gains, with a slight super-linear improvement in performance of 28% (improvements due to increased cache). `Swim` only exhibits a 5% speedup with *2f1s-G*, and even *3f* only improves performance by 12%, which indicates the processors are often idle waiting for memory. Running with more threads results in increased off-chip memory pressure, which is why `swim` fails to improve in performance. We next examine average per-benchmark power to confirm this.

4.3.2 Power and Energy

Activating a third processor at the same voltage but lower frequency yields less energy savings than lowering voltage in tandem with frequency. This is due to the quadratic coefficient voltage plays in the power equation ($\frac{1}{2}CV^2f$). There should be a 25% net increase in CPU power consumption for computation intensive benchmarks. Actual energy consumption varies since dynamic power consumption for various programs

(L3 cache accesses, memory controller, main memory functions, off-chip accesses, and cache coherence) are amortized across all cores.

Figure 4.6 graphs the power consumption of the three processor (*2f1s* and *3f*), and four processor (*2f2s*) cases normalized to the two processor case. *2f1s*'s power consumption is about half way between the power consumption of the *2f2s* and the baseline configuration for most benchmarks. The *3f* configuration generally has the highest power consumption. A workload imbalance reduces throughput, but does not significantly reduce power consumption of the cores for the *2f2s* and *2f1s* configurations. Although *lu* shows performance degradation for *2f1s* and *2f2s,* power consumption increases by 12% and 27% respectively. `Ep` power consumption with the *2f1s* configuration is 30% higher than the two thread case, slightly higher than the theoretical increase of 25% for computation bound applications. The higher consumption can be attributed to increased power consumption of shared structures, which consume power at a rate independent of the slow core, such as cache coherence. `applu` exhibits a slight increase in performance (3%) and decrease in power power consumption (7%) with the *2f1s* configuration. The slower thread for *2f1s* slows down the others, such that increasing numbers of threads does not improve performance, but reduces power consumption of the faster running processors. This is proven from the instruction mix and performance graphs seen earlier. `applu` exhibits a 20% improvement in performance with no increase in power with the *2f2s* configuration. This would not be noteworthy if we were reducing frequency and voltage when increasing threads. However, here we increase the numbers of threads without reducing the frequency or voltage from cases using threads. This indicates that for the two thread configuration, the threads are performing inefficiently. For `lu`, *2f1s* and *2f2s* are also inefficient configurations, where power increases by 12% and 27%, while performance actually degrades. This is because processors are still spinning on locks, executing useless instructions while stalled at barrier points. Overall, *2f1s* and



(a) NAS (b) SPEC-OMP

Figure 4.6: Power Consumption Normalized to Two Thread CMP (Lower is Better)

2f2s use 12% and 26% more power, respectively, on average, during execution. This is lower than our theoretical maximum, but accurately matches the trends in performance improvement seen above. The exception to this is `lu`.

Using more threads requires more power, since voltage or frequency do not decrease. However, if the benchmarks achieve a significant performance improvement, total energy is decreased due to power being consumed for less time. This is because shared structures are active for less time, and the dynamic power consumption is amortized over greater threads. We compare total energy consumption of the *2f1s*, *2f2s*, and *3f* configurations with the two-core homogeneous CMP. Figure 4.7(a) graphs energy consumption for *NAS*, and Figure 4.7(b) shows energy consumption for *SPEC*. Results are normalized to the baseline homogeneous case. To attain these energy savings, we assume the system can enter a low power state or start (be turned off or on) on the next job, when idle after completing the previous job. With the NAS benchmarks, energy reduction varies, depending on performance improvements and increased power consumption. `Ep` requires slightly more energy with more threads, since the improvements in performance closely correlate with increases in power consumption. `Cg, ft`, and `sp` show highest energy reductions with the heterogeneous configurations. For these benchmarks, improvements

from increasing threads, and improvements from reduced cache misses leads to greater efficiency. This is especially true for `cg`, which is why it has the highest energy reductions with more threads. `Lu` shows significant increases in energy consumption due to the decrease in performance and increase in power consumption from using heterogenous configurations. For `lu`, the homogenous configuration does not improve over the baseline, since improvements in performance are offset by similar increases in power consumption.

All SPEC benchmarks except `swim` show energy reductions for all configurations. Recall that `swim` is memory constrained and suffers from many memory stalls, thus running extra threads without decreasing voltages or frequencies fails to improve energy efficiency. The worst degradation is less than 6%. However, even though processors are stalled waiting for memory, their power consumption is low. *3f* is the most inefficient for `swim`, another indication that high frequencies do not help. From 4.7(b), it is clear the heterogenous configurations are just as or more energy-efficient than the homogenous configurations. The exception is `applu`, where workload imbalances hampers energy reductions, and efficient operation. When instructions are stalled due to memory latency, processors attempt to issue other instructions by reordering instructions to execute. This leads to extra work that might not be done at lower frequencies, where memory from the processor's perspective has a lower latency. This is one of the reasons why memory bound benchmarks have higher energy savings with the *2f1s* configuration compared to the *3f* configuration.

Execution times in seconds are shown for NAS and SPEC-OMP benchmarks in Tables 4.3.2 and 4.3.2 respectively. Thread configurations are listed from *2f* to *3f*, with *s* denoting a slow core (50% maximum frequency) and *f* denoting a fast core (maximum frequency); the preceding number indicates the number of cores running at that speed. *2f*

Table 4.4: NAS Execution Time in Seconds (Lower is Better)

APPLICATION	2f	2f1s-S	2f1s-G	2s2f	3f
bt	236.2	306	202.71	175.08	167.03
cg	93.8	100	68.21	57.55	65.04
ep	65.81	87.7	52.48	43.78	43.87
ft	55.38	65.7	43.78	38.12	37.48
lu	283.53	305.3	351.29	341.22	231.04
mg	9.92	11.2	8.64	8.16	7.93
sp	210.04	237.9	179.71	159.5	160.67

Table 4.5: SPEC-OMP Execution Time in Seconds (Lower is Better)

APPLICATION	2f	2f1s-S	2f1s-G	2s2f	3f
applu	67.68	70.23	65.52	56.18	45.14
apsi	798.9	1001.12	701.21	513.12	513.57
equake	39.2	40.16	30.62	27.66	29.03
mgrid	103.95	132	87.61	74.95	72.88
swim	66.99	69.91	63.73	61.64	60.06
wupwise	1303.89	1601	1039.85	880.44	903.79

is the two-thread configuration used as our baseline. *2f1s-S* and *2f1s-G* are three thread configurations, with *2f1s-S* for static scheduling and *2f1s-G* for guided scheduling.

Energy consumed in joules is listed in Tables 4.3.2 and 4.3.2 for NAS and SPEC-OMP benchmarks respectively. Energy data for the *2f1s-S* configuration is omitted since it is not competitive with other thread scheduling methods. Configurations *2f, 2f1s* (using guided scheduling), *2f2s* and *3f* are shown, using the same label notation as the performance graphs.

We examine cache activity from switching a homogenous two thread configuration to a heterogenous four-thread configuration. Figure 4.8 shows memory behavior for the four thread *2f2s* case normalized to the two thread *2f* case. Individual L2 cache miss

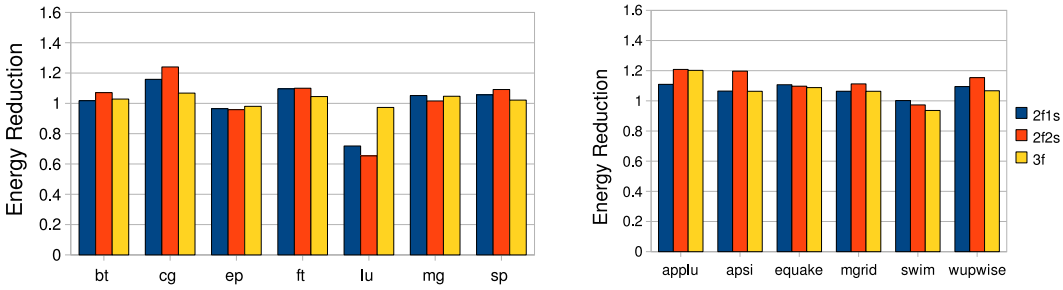
Table 4.6: NAS Energy Consumed in Joules (Lower is Better)

Application	2f	2f1s	2s2f	3f
bt	12991	12768.27	12133.04	12633.89
cg	5327.84	4599.26	4295.48	4990.7
ep	2303.35	2387.34	2403.72	2349.68
ft	3134.51	2860.14	2850.11	3000.16
lu	16359.68	22764.29	25040.71	16812.01
mg	610.08	580.29	600.58	582.78
sp	11810.55	11182.09	10823.34	11562.66

Table 4.7: SPEC-OMP Energy Consumed in Joules (Lower is Better)

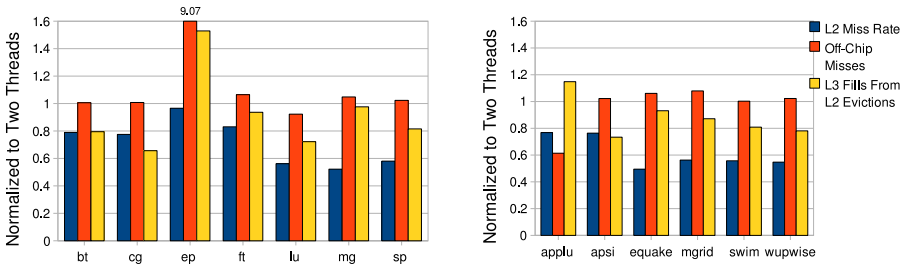
Application	2f	2f1s	2s2f	3f	
applu	3477.74	3134.48	2877.82	2892.65	
apsi	35751.48	33575.66	29871.34	33615.05	
equake	2088.38	1886.69	1903.01	1919.71	
mgrid	6215.71	5843.59	5586.97	5844.29	
swim	4181.28	4173.61	4296.91	4466.36	
wupwise	75666.69	69127.89	65578.41	70926.21	

rates, total off-chip misses and L3 cache fills from L2 evictions are graphed. Reductions in L2 cache misses improve performance and reduce energy consumption. Since off-chip misses do not increase with increasing threads, the off-chip power is amortized over more cores, so efficiency (defined as performance per watt) increases. The reductions in L3 fills from L2 evictions reduces L3 cache accesses, reducing energy consumption. The benchmarks exhibit reductions in L2 miss rates and L3 fills from reduced L2 misses with increasing numbers of threads. `Ep` has significant increases in off-chip misses (900%) with increasing numbers of threads. However, actual values are very small, with a negligible effect on performance. While these reductions improve performance and energy, their effects on energy reduction will vary, depending on the ratio of microarchitecture activity to cache accesses. We do not graph L2 cache



(a) NAS (b) SPEC-OMP

Figure 4.7: CMP Energy Reduction Normalized to Two Thread Configuration (Higher is Better)



(a) NAS (b) SPEC-OMP

Figure 4.8: *2f2s* vs. *2f* Cache Behavior (Lower is Better)

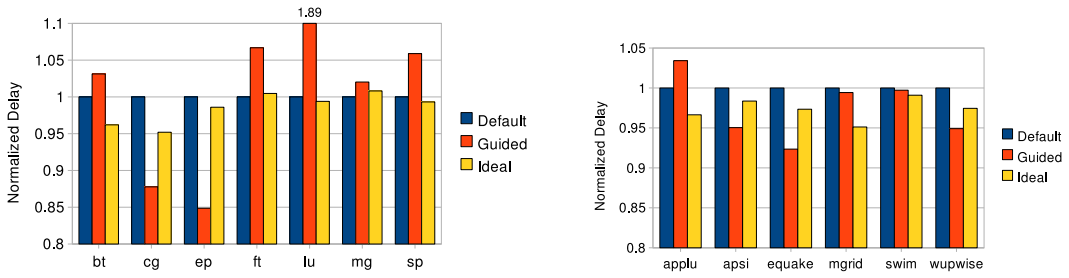
coherence snoops, which should grow with increasing numbers of threads due to our broadcast based cache coherence. However, L2 snoops only require checking the cache tags, using significantly less energy than a cache miss that entails line fills and off-chip accesses. While the reduction in L2 misses is not as significant as it was for `cg` going from one to two threads, we find using the most threads can be beneficial, due to non-uniform program scaling from a reduction in memory footprint size.

A scheduler might be inclined to clock down one of the cores on a CMP and inherit this "heterogeneous" configuration, such as with *2f1s*, trading off performance for power and energy reductions. Alternatively, one can tradeoff die area for energy reductions with the *2f2s* configuration.

4.3.3 Sensitivity to Performance Variation

We examine scheduling for small variations in CPU frequency. We assume we have a four-core CMP with a target frequency of 2GHz, with a 10% process variation. This results in a configuration with one core running 10% slower (at 1.8 GHz) and another core running 10% faster (at 2.2 GHz). We use the Windows platform to choose frequency increments of 100 MHz from 1.0-2.2 GHz.

Figure 4.9 shows performance for different scheduling methods for the (a) *NAS* and (b) *SPEC-OMP* benchmarks. *Default* configuration represents the benchmarks run with default compilation parameters with static scheduling. *Ideal* configuration shows benchmark performance on a frequency equivalent four-core CMP with homogeneous frequencies (2 GHz per core). We show performance as delay, normalized to the default case, where lower delay is better. Most benchmarks do not significantly suffer from running on the heterogenous cores since the frequency variation is small, and the benchmarks are limited more by memory than frequency with four threads at 2 GHz. Therefore, even when the slower core lags behind the faster processors in operating frequency, it is not a performance bottleneck that results in other threads stalling at barrier points. The guided configuration represents benchmarks run with guided scheduling. *Guided* performs badly for more than half the NAS benchmarks, most notably `lu`, where delay is increased by 89% due to a large workload imbalance. Some benchmarks, such as `cg` and `ep`, show better performance with *guided* rather than with the *ideal* case. For *SPEC-OMP*, the benchmarks are not constrained by memory, and performance degrades noticeably between the ideal and default configurations. The guided configuration is able to improve performance for most cases except `applu` where delay is increased by 3%. Some benchmarks, such as `apsi` and `equake`, exhibit improvements better than the *ideal* configuration by using *guided* scheduling, due to better run-time workload bal-



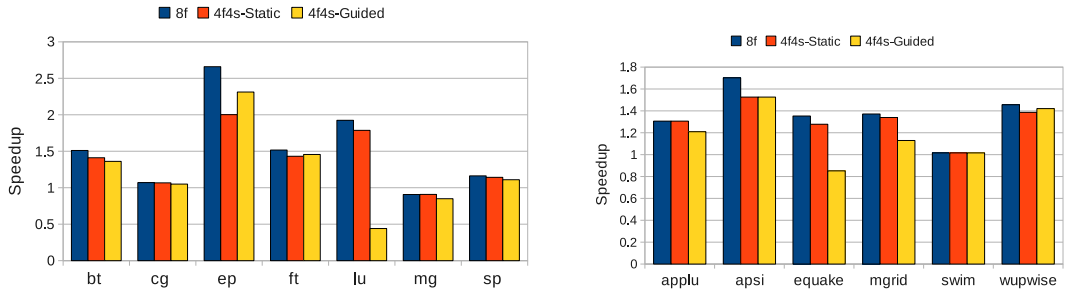
(a) NAS (b) SPEC-OMP

Figure 4.9: Scheduling Delay Normalized to Default Static Scheduling (Lower is Better)

ancing. `Swim` is heavily constrained by memory, making frequency and even reducing thread counts irrelevant to performance.

With a 10% process variation, the performance differences between *static* and *ideal* scheduling are negligible for NAS, making *guided* scheduling redundant and even detrimental to performance for some benchmarks. SPEC-OMP benchmarks are more sensitive to frequency, and their performance is improved through the use of guided scheduling. Actual improvements vary depending on application dependence on frequency and thread count.

Guided scheduling improves performance over *static* scheduling for non-memory bound benchmarks. For memory-intensive benchmarks, the performance variation between fast and slow processors is negligible and the overhead of *dynamic* scheduling eclipses the benefits. We repeat our experiments for an eight core system, and find similar trends. Figure 4.10 graphs speedup via non-uniform scheduling, normalized to the *static* performance of four cores, on a homogenous 2.66 GHz eight core system with a 25% variation in frequency between fast and slow cores. The 4f4s-Static represents the normalized delay of using static scheduling on an eight-core system composed of four fast cores at 2.66 GHz, and four slow cores at 2.0 GHz. The *4f4s-Guided* represents



(a) NAS (b) SPEC-OMP

Figure 4.10: Speedup Normalized to Four Cores (Higher is Better)

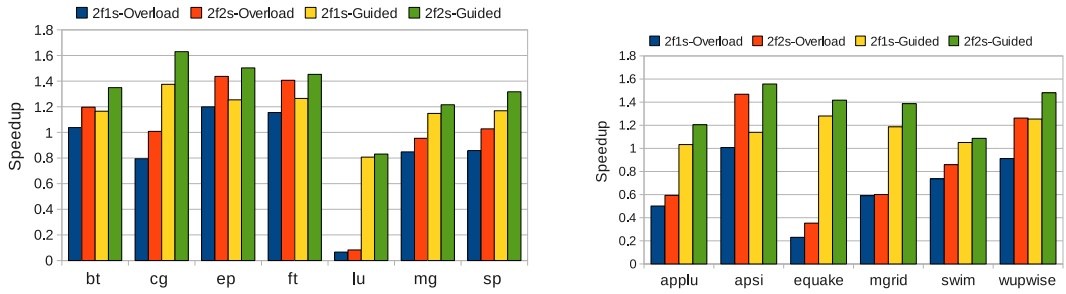
a similar configuration, but using *guided* scheduling. For the eight core case, *guided* scheduling can severely degrade performance, in which case it is better to use normal *static* scheduling than to try to balance the workload at run-time. Most benchmarks are sufficiently memory bound that the frequency heterogeneity does not change their overall efficiency with static partitions. The most benefit of *dynamic* scheduling is extracted from computationally bound code that is executed on a substrate with large variations.

4.3.4 Virtual Thread Performance

We compare the performance of the OpenMP workload balancing technique against increasing the number of virtual threads. Increasing the number of threads also breaks workload chunks into smaller segments, which can offset the imbalance in speed among processors. This is also useful for instances in which a program can only execute a certain number of threads or does not perform loop partitioning. Since, by design, OpenMP scheduling only works for loop partitioning, if a specific thread count is required, the code likely does not parallelize loops, and OpenMP dynamic workload balancing cannot be used. While none of our programs exhibits preferences for specific thread counts, we evaluate overloading processors with more than one thread to meet thread count require-

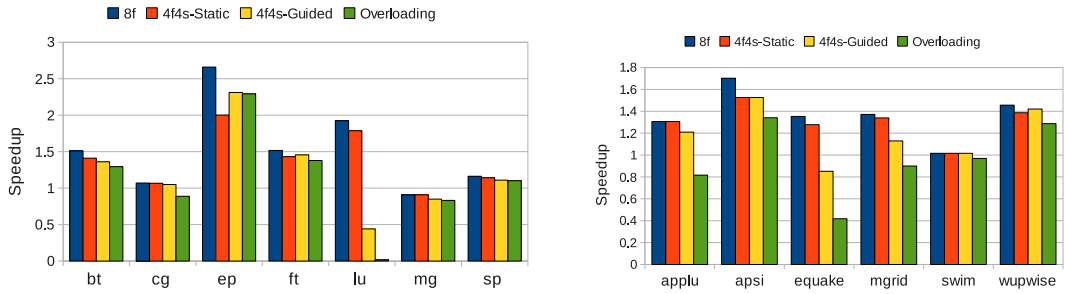
ments, and compare this against *guided* scheduling. The results for the four core case are shown in Figure 4.11 for *NAS* and *SPEC-OMP* benchmarks. Overload denotes using four threads per core, and *2f1s* denotes two fast cores operating at 2.2 GHz and one slow core at 1.1 GHz. Performance is normalized to using two fast cores with *static* workload partitions, where higher is better. *Guided* scheduling does better than overloading threads on cores since there is far less thread contention. *Overloading* processors with extra virtual threads degrades performance for memory dependent applications due to cache contention and synchronization at barrier points. Programs that are computation bound, such as `ep`, benefit from virtual threads, but it is not as efficient as using the OpenMP scheduling methods.

Figure 4.12 graphs speedup for the eight core case, with four fast processors operating at 2.66 GHz, and four slow processors at 2.0 GHz. *4f4s-Static* denotes static partitioning and *4f4s-Guided* denotes guided scheduling. *Overloading* is using more virtual threads than processors, and all results are normalized to four cores operating at 2.66 GHz. The results for eight-cores running at maximum frequency are included to provide an upper bound for the speedup, shown as *8f*. We include our earlier dynamic scheduling results for comparison. Compared to the four core case, the degradation at eight cores is even higher, such as for `lu`. Depending on speedups from using the extra cores, overloading is another option if OpenMP dynamic partitioning cannot be used for a specific thread count, but performance needs to be monitored to ensure that overloading is not degrading performance over using fewer cores.



(a) NAS (b) SPEC-OMP

Figure 4.11: Four Core Performance with Virtual Threads



(a) NAS (b) SPEC-OMP

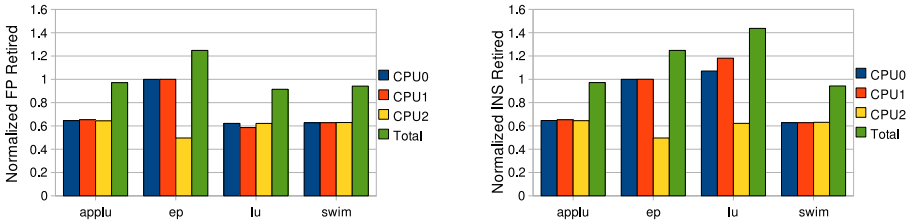
Figure 4.12: Eight Core Performance with Virtual Threads

4.3.5 Metrics for Run-time Scheduling

In Section 4.4, we find that increasing numbers of threads does not always improve performance, and even when it does, it might not be energy efficient. It would be useful to determine an efficient number of threads in real-time without offline profiling for frequency-heterogeneous CMPs. To achieve this, we need to be able to determine thread progress and application power consumption. We examine these using performance counters to determine useful thread progress and whether increasing thread count improves performance. We use power meters to determine real-time power consumption of an application for a given configuration. We specifically examine total instructions

retired, and total floating-point (FP) instructions retired (inclusive of MMX instructions) per processor to gauge thread progress. We examine whether using either of these performance counters is sufficient to determine thread progress. The drawback of the total instructions retired metric is that stalled threads spinning on locks can lead to increased instruction counts without increasing work throughput. Unfortunately, OpenMP does not support dynamically changing the number of threads via signals sent outside the program, which prevents us from implementing a real-time scheduler that detects whether increasing the thread count reduces performance. Our results are thus based on static thread counts, but nothing precludes this work from being done in real-time, should the underlying shared-memory framework support it.

Figure 4.13(a) graphs floating-point instructions per core per second for `applu`, `swim`, `lu`, and `ep` for the *2f1s* configuration, normalized to the two-thread homogeneous CMP configuration. We choose these benchmarks since they suffer the worst degradation from increasing thread counts, except for `ep`, which we use as a point of comparison. For the third core, we normalize retired FP instructions to one of the homogeneous cores (which of the two cores we choose does not matter, since both retire approximately the same number of FP instructions) from the two-thread runs. The *total* column represents total FP throughput normalized to the homogeneous two-thread case. Ideally, for computation bound cases there should be no change in cores 0 and 1 when thread counts increase, and 50% throughput for core 2. This is the case for `ep`, which is why it has a 25% increase in total FP instructions retired. Other benchmarks show large degradations for cores 0 and 1 when thread counts increase, resulting in fewer total FP instructions retired than for the two thread case. Optimal thread counts are chosen dynamically (without profiling). Interestingly, core 2 shows throughput higher than 50% for many benchmarks, indicating programs are bound by memory and not by frequency. We do not use total instructions committed to gauge throughput, since all



(a) Floating-Point (FP) (b) Integer (INT)

Figure 4.13: CMP Retired Instructions Scaling from 2 to 3 Threads

of the benchmarks we evaluate are FP, and processors, spinning on locks may increase integer throughput without improving performance. This phenomenon is illustrated for `lu` in Figure 4.13(b), which graphs the total instructions per core per second. Results are normalized to the two thread homogeneous CMP configuration. While other benchmarks follow the trends from Figure 4.13(a), `lu` shows how using the total instructions retired metric could lead to erroneous thread choices, since it exhibits higher instruction throughput at higher thread counts. Processor stalls are not a good metric to detect stalls for applications limited by memory or barriers such as `lu`, since cores are not stalled when spinning on locks. These results can be used for sampling the entire search space at run-time until the optimal configuration is found (although this may not be feasible for an OS scheduler).

For energy consumption, a power meter can be used to determine system power, or if only some of the cores are being used within the system, it can be estimated using the method we used in Chapter 3 via performance counters [49]. Knowing power consumption combined with determining performance throughput using performance counters enables us to determine energy consumption.

4.4 Conclusions

We investigate performance and energy improvements from leveraging cores that operate at different frequencies on real hardware. By balancing workload chunks between faster and slower cores, we reduce bottlenecks caused by slow cores. We find dynamic scheduling methods can add noticeable overheads and affect performance. The guided scheduling method with a single lower bound for chunk size works well for most benchmarks, while dynamic scheduling with fixed chunk sizes does not. Workload balancing is imperfect, with a variation of 5% above ideal on average for a quad-core CMP. With three heterogeneous cores, this yields a 25% load imbalance on the slowest core.

For our quad-core CMP with large frequency variations, we observe speedups of up to 38% and reduce total CPU energy consumption by up to 16%. While increasing the number of threads increases power consumption, reducing execution time yields overall reductions in energy consumption. We find running memory-limited benchmarks in heterogeneous configurations to be competitive with homogeneous configurations at identical thread counts, but with greater energy reductions. In this case, the heterogeneity of the cores is masked by uniform memory speeds.

On average, the heterogeneous three-core system uses the same energy as the homogeneous three-core system, indicating heterogeneous cores are not as inefficient as originally thought. We find varying improvements, depending on the benchmark, but the variation results more from program scaling and memory bottlenecks than from the underlying heterogeneity of the CMP. The clear exception to this is *lu*, which suffers from a gross workload imbalance among cores and high overhead from dynamic scheduling. Although not emphasized in previous work by Balakrishnan et al. [4], the memory bottleneck of CMPs plays a large role in the scalability of heterogeneous systems. These

systems perform almost as well as their homogeneous counterparts, as memory is independent of the CMP's asymmetry. This is verified by the homogeneous higher thread count tests not performing noticeably higher for some benchmarks. Although in the majority of cases we see an improvement in performance with increasing numbers of threads, `lu` shows significant degradation from multithreading in a heterogeneous environment. This is due to the overheads of scheduling and the unbalanced workloads. Therefore utilizing under-performing processors can result in reduced energy consumption and improved performance, if the scheduler can manage the workload distribution by scheduling in a transparent manner, little code modification (if any) is required by the programmer. When evaluating frequency heterogeneity for an eight-core system with modest frequency variation, we find little improvement from workload balancing. A program's failure to scale, as well as memory limitations, reduce the potential for improvement over the homogenous case. Additionally, these two factors also reduce the heterogeneity between fast and slow processors. Hence, actual performance gains on frequency heterogenous systems are dependent on the application's reliance on memory, as well as the degree of variation between fast and slow cores.

We find using OpenMP pragmas to be better at workload balancing than overloading cores with extra virtual threads. Only in cases where an application is compute bound do we observe improvements using virtual overloading rather than using no workload balancing techniques. Virtual threads, however, are a viable alternative for parallel programs where OpenMP pragmas are not used or certain thread counts are required, such as powers of two or even thread counts.

We describe a feasible method to detect at run time whether increasing thread counts will improve performance. Combining this with our work on estimating power consumption [49] enables us to determine whether using extra cores is more energy effi-

cient. As the number of on-chip cores grows, significant inter-core heterogeneity will be inherent. Managing this will be crucial in determining the future processor landscape.

CHAPTER 5

CO-SCHEDULING FOR FREQUENCY HETEROGENEOUS CMPS

5.1 Introduction

Co-scheduling leverages the different concurrency points at which multithreaded programs are most efficient, to improve overall throughput and reduce energy costs. In Chapter 3, the scheduler only needed to choose what programs and threads should be executed in unison. Scheduling for a heterogeneous CMP adds another dimension to the problem: to which core to assign the threads. However, this also gives the scheduler finer granularity in making scheduling decisions, since the core types chosen allow badly scaling programs to retain faster processors, and discard slower processors (by the scheduler). Memory intensive applications can use slower processors, and faster processors can be retained for applications that exhibit the best performance on those processors.

We leverage our previous work on the fair scheduler to create a heterogeneous processor aware scheduler. A processor-aware scheduler has the potential to achieve better performance by mapping applications to processors on which they have the best performance, defined as instructions retired per clock cycle. When applications are limited by memory rather than computation, slower processors can be more energy efficient [27], reducing energy consumption with negligible performance degradation.

Since we perform run-time adaptation based on application behavior, we need to sample application performance during execution. We increase our sampling stages by two to use four time quanta for initial sampling of each of the programs. Two time quanta are used to sample the performance of the program at the highest and lowest

thread counts as done previously in Chapter 3, and two time quanta are used to sample the program separately on the fast and slow cores. IPC levels indicate how efficient the program is on processors of different frequencies. Programs that are not bottlenecked by memory would have the same IPC on all cores, regardless of frequency. Programs are that are limited by memory have lower IPCs on faster processors, indicating they are better suited for slower cores. The scheduler can use this information to decide what programs benefit most from the faster cores. If the unassigned processors are all homogeneous, than this information is not used. IPCs are calculated using floating point operations to ensure synchronization overhead is not affecting IPC counts. However, performance counter data among synchronization points can also be used for integer programs to ensure that the data used for calculating thread progress does not reflect erroneous information [16]. We define *processor affinity* in equation 5.1, which quantifies how much an application prefers a faster processor over the slower one. The higher the processor speed affinity, the higher the probability that the scheduler will give that application faster processors.

$$Processor\ Speed\ Affinity = \frac{IPC\ faster\ core}{IPC\ slower\ core} \tag{5.1}$$

We modify the *fairness* heuristic equations from chapter three to account for processors operating at different speeds (see Equations 5.3- 5.6). If a program is scaling from only using fast cores to also using slower cores, then its efficiency level needs to account for the new cores being significantly slower. We use the fastest processors as our generic baseline processor, and convert slower processors into a uniform metric. In our study, the slower processors operate at 25% lower frequency than the faster processors, resulting in each slow processor being equivalent to 75% of the faster processors. We quantify this in Equation 5.2, which shows how processors are converted to a generic

representation, based on their operating frequency (represented in MHz). P_g represents the processor converted to a generic format. *HF* represents the highest frequency in our system, in MHz. In our test machine, the fastest processor operates at 2666 MHz, so 2666 is used as a constant for our experiments, and is substituted into Equation 5.2 for *HF*.

$$P_g = \sum_{i=1}^{N} \frac{physical\ frequency}{HF} * P_{physical} \tag{5.2}$$

This new generic processor representation is used to augment our previous *fairness* and thread throughput equations, resulting in two new equations: fairness Equation 5.3 and thread throughput Equation 5.4. For the thread throughput equation, the generic thread represents a generic CPU, also based on Equation 5.2.

$$Fairness\ =\ \frac{New\,Cores_g}{Old\,Cores_g} * \frac{New\,Time\,Quantum\,Size}{Old\,Time\,Quantum\,Size} \tag{5.3}$$

$$Normalized\,Thread\,Throughput\ =\ \frac{Number\,of\,Instructions\,Retired}{Threads_g\ *\ ExecutionTime} \tag{5.4}$$

Equations 5.5 and 5.6 remain the same as in Chapter 3, since the data they require already encompass the generic value for the processor. Conditions 5.7 and 5.8 dictate when processors can be deallocated from an application, and indicate what the largest value for $alpha$ can be (adapted to handle non-uniform processors).

$$Thread\,Throughput\,Gain\ =\ \frac{Throughput\,With\,Co-scheduling}{Max\,Thread\,Throughput} \tag{5.5}$$

$$Speedup =\ Fairness * (Thread\,Throughput\,Gain) \tag{5.6}$$

The application chooser phase is augmented slightly from Chapter 3. Applications are still initially assigned processors round-robin, in an attempt to assign equal numbers

of cores whenever possible. However, the actual processors assigned to a thread are based on the thread's CPU *speed affinity*, where an application is given the processor to which it has the highest affinity, compared to its co-scheduled neighbors. This augments our prior application pairing and thread count choosing heuristics.

The application chooser follows the following steps.

1. It chooses the highest resource usage program (head of list B) and adds it to co-scheduling list C.
2. It chooses the programs with the lowest resource usage (head of list A) from our set of programs, adds it to list C and removes it from list A. *Cores are distributed round-robin to each of the applications, with actual cores allocated based on the CPU affinity of the program receiving the allocation.*
3. If the ED for all applications co-scheduled are less than what they were before co-scheduling, then the application most recently added to list C is removed from the head of list A, and *Step 2* is repeated. If the ED for any application is worse after co-scheduling, and this is the first failed co-scheduling attempt, than it proceeds to the next step. If this is the second time co-scheduling has failed, it proceeds to *Step 5*.
4. The program at the head of list C is removed and returned to the head of list A. The second program in list A is added to list C. It divides the number of cores round-robin (sorted by lowest resource use) until no more unassigned cores remain. *Step 3* is repeated.
5. The program at the head of list C is removed and returned to the head of list A. The programs remaining in list C are sent to the thread chooser, and the application chooser repeats starting from *Step 1* with the set of programs remaining in lists A and C.

The thread chooser is mostly left unchanged. However, in instances where a thread is added or removed from an application's processor set, and the set consists of heterogenous processors, the slowest processor's thread is chosen first. The thread chooser follows these steps:

The thread chooser follows the following steps:

1. It records the throughput of each of the programs at the equivalent thread counts previously chosen by the application chooser. For two programs, it would be the throughput for equally dividing the number of cores among benchmarks. Therefore, if N is the total number of cores available, it would be the throughput where each process receives $N/2$ cores. However since the cores are not homogeneous, the partition is not necessarily equal. Equation 5.3 is not applied in this step.
2. It increases and decreases thread counts for each program within the list, alternating between applications. This creates two lists, a list containing the reduction in ED of each application from increasing its thread count, and a list containing the increase in ED of each application from decreasing its thread count. This requires two time quanta if the list contains an even number of applications, and three time quanta if the list contains an odd number of applications. Generally, a program will scale differently, depending on which of its neighboring applications donated the thread it is using for scaling up. We make the simplifying assumption that applications scale up independently of other programs its co-scheduled with, so the thread chooser does not need to sample *C(A,A/2) (A choose* $A/2$*)* different combinations.
3. Threads are increased from applications in list A, and decreased from applications in list B. The process repeats until the applications in list B show worse ED than the baseline case of gang-scheduling, or the overall ED from selectively increasing

application threads decreases. Since the process could potentially continue for N iterations with N cores, we limit the number of iterations to two to reduce number of time quanta required for a solution.

$$\frac{Ir_{t+1}}{Ir_t * (1-\alpha)} < \frac{(N+1)_g}{N_g} \tag{5.7}$$

$$\alpha < 1 - \frac{N_g}{(N+1)_g} \tag{5.8}$$

This processor-aware scheduler maps applications to cores based on sampled performance of applications on cores and each application's use of shared structures. To accomplish this, it marginally increases the number of time samples needed by two. If heterogeneity exists at more than two frequency points, performance at the highest and lowest frequencies can be used to extrapolate performance at other points, so that further sampling is not required. However, if the heterogeneity is architectural, rather than solely frequency based among processors, performance data on each of the different processors is required. Our heuristic scheduler can achieve efficient solutions without searching the entire design space because it does not guarantee the optimal solution, but an efficient schedule compared to the baseline.

Since in our experimental infrastructure, at most half of the cores are crippled (either from variation or from thermal constraints), the other full-frequency cores can be clocked down in instances where high frequency operation is energy-inefficient or does not improve performance. We do not investigate frequency throttling for the faster processors because there is little potential for improvement on our high performance system evaluation. Very few of the benchmarks in the SPEC OpenMP suite are sufficiently limited by memory that processor frequency can be reduced without degrading perfor-

mance. Additionally, there are negligible energy improvements from frequency scaling because the difference in dynamic power consumption is not significant compared to the total power consumption of the entire system. Similar to Amdahl's law for performance, the potential for energy improvements from solely reducing the power of a set of four cores is low in comparison to the total power of the system. We illustrate this quantitatively using our evaluation system as an example. The maximum observed difference in power consumption between keeping four cores at 2.0 GHz versus 2.66 GHz is 20W. While this is not trivial, since our test system's idle power consumption is 180W. The difference in system power consumption between clocking down half the cores or running at max frequency is less than 10% of total system power once the system is actively executing a multithreaded application. This 10% reduction in power does not result in 10% energy savings, since benchmarks exhibit an increase in delay. Only two of our benchmarks show less than a 10% performance degradation when switching from fast to slow cores (`swim` and `applu`). Additionally, unlike giving up space for time, reducing frequency does not increase an application's time quantum, reducing the performance incentive for an application to use slower processors.

5.2 Experimental Setup

Our test system is a 2.66 GHz eight-core system consisting of four dual-core CMPs in an SMP configuration, each dual-core CMP sharing a six MB L2 cache, and having private L1 caches. Each chip consists of two dual-core CMPs, and each chip (of four cores) can be configured to run at either 2.0 GHz, 2.33 GHz or 2.66 GHz. Our heterogenous configuration consists of four cores at 2.66 GHz and four cores at 2.0 GHz. Table 5.2 lists system configuration parameters. We use RedHat Enterprise Linux 4, running kernel 2.6.28.1 with SMP support. Our kernel is modified to support real-time temperature

Table 5.1: 8-Core SMP Machine Configuration

Frequency	2.66, 2.0 GHz
Process Technology	45 nm
Processor	Intel Xeon E5430 CMP
Number of Cores	8, dual-core SMP
L1 (Instruction) Size	32 KB 8-Way Set Associative
L1 (Data) Size	32KB 8-Way Set Associative
L2 Cache Size (Shared)	6 MB 16-Way Set Associative
Memory Controller	Off-Chip, 4 channel
Main Memory	8 GB FB-DIMM (DDR2-800)
Front Side Bus	1333 MHz

and performance counter data parsing from processor sensors. We use *pfmon* from the *perfmon2* package to read the on-chip performance counters.

We use performance counters to estimate power consumption for different cores on chip, and to calculate power consumption of each program a multiprogrammed workload. Using the method outlined in our earlier work here [49], we find very low median error rates of 3%, when verifying this method with the SPEC2000, SPEC2006, SPEC-OMP, and NAS benchmarks. We also verify these estimates using a Watts Up [18] power meter on our heterogenous eight-core system. Our power model can estimate power consumption for the system running with all cores at the same frequency or at different frequencies.

We use the SPEC-OMP shared-memory benchmark suite [2] to evaluate our scheduling policies. We modify the code to use run-time scheduling, which we found in Chapter 4 to work well with most workloads. We compile with `-openmp` and `-static` flags using Intel C and Fortran compilers. We use medium reference inputs and run our benchmarks natively. Galgel fails to compile for our platform, and so is omitted

from our analysis. We use the Linux *taskset* command for binding our programs to specific processors, ensuring they do not migrate to non-assigned cores. We use the longest stable period from our suite of programs as our time quantum for measuring program performance. Each time quantum is three seconds long, and amortizes the time to swap programs on the CMP. Shorter running programs are looped to keep executing until longer running programs finish. This prevents the different run lengths of programs from skewing performance results. We choose the SPEC-OMP benchmark suite because it contains many programs that fail to scale and have a diverse set of memory requirements. Additionally, they all use OpenMP pragmas to partition their workload, which enables us to use the dynamic workload balancing techniques we explored in Chapter 4 to adjust for and retain performance when migrating programs to a heterogeneous substrate.

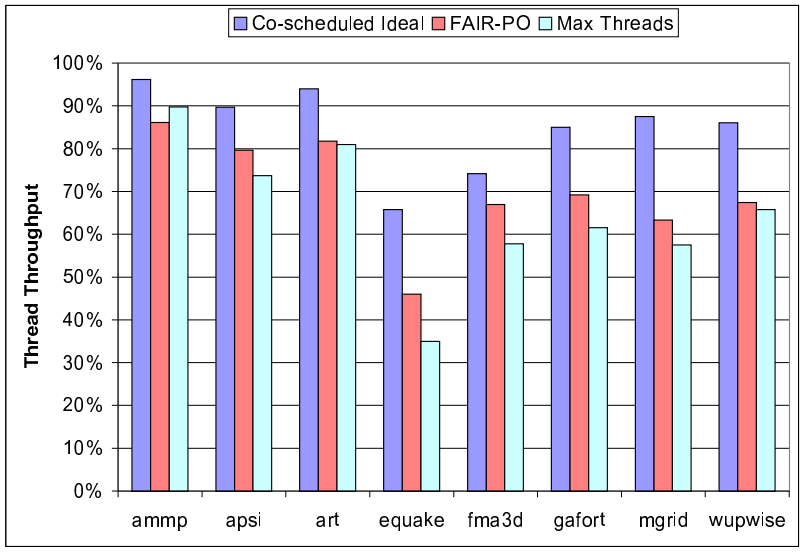
5.3 Evaluation

We use the best case time-shared gang-scheduler as the baseline; it distributes the workload between fast and slow cores using either static or dynamic scheduling (whichever offers better performance). We find using guided scheduling achieves the best performance for all programs except one (`mgrid`), for which we use static scheduling. We evaluate our prior substrate-oblivious HOLISYN *fair* scheduler, as well as our new scheduler that is processor aware.

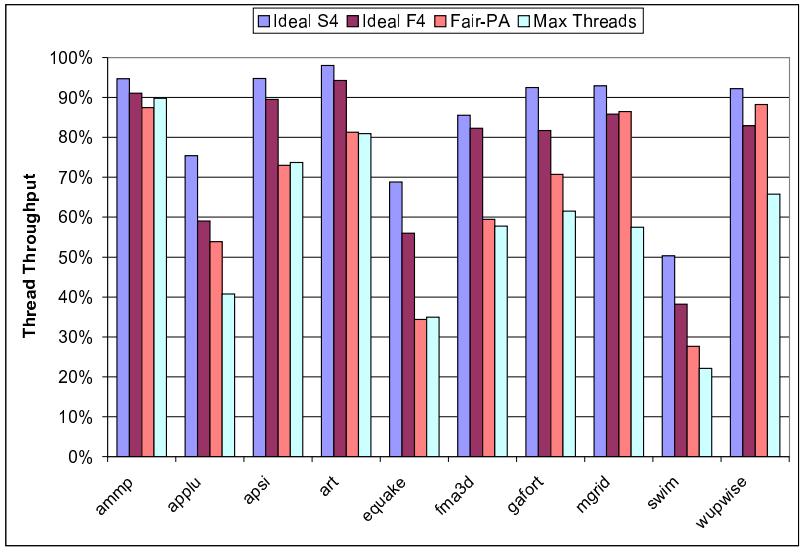
We examine the other extreme of scheduling, which is concurrently scheduling all applications, and not using time-scheduling to swap applications. We attempt scheduling in a resource-oblivious but processor-aware manner. This strategy runs all applications concurrently, but maps applications to cores on which they have the best per-

formance, based on IPC. We map threads to cores on which they have the highest IPC normalized by frequency, and call it round-robin IPC (RR-IPC) scheduling. Therefore a memory-intensive program is mapped to a slow core, and a compute-intensive program is mapped to a fast core. This is in the spirit of scheduling for heterogenous architectures, which has previously been advocated by researchers for single-threaded applications [62] [35], but adapted for multithreaded applications. De Vuyst et al. [62] find improvements by scheduling threads on SMT cores for which benchmarks exhibit the highest IPC. Kumar et al. [35] also advocate mapping processes to cores on which they exhibit the highest IPC, normalized by processor chip area; however all of our cores have the same chip area, so area is factored out. If the application is memory bound, slow cores will have higher IPC than faster cores, otherwise the latter cores will have equivalent IPC to slower cores. We omit running `gafort` and `apsi` concurrently, since we have more applications than cores, and their memory requirements prevent them from running concurrently with other benchmarks. Running them with other benchmarks results in their never completing due to excessive disk swapping. However, this highlights one of the issues with this scheduling method: it requires enough main memory to hold all of the programs simultaneously, which may not be feasible, depending on the application inputs used. Increasing memory is one option, but it will increase system power without any necessarily tangible improvements.

We compare our different schedulers with the oracle scheduler, which creates the best schedules by optimizing for ED, while having the least performance impact on all the programs.

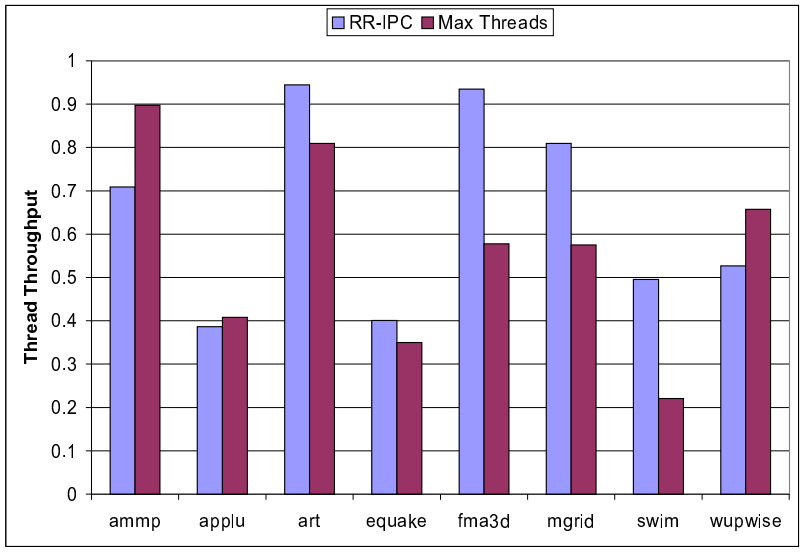


(a) Fair-PO Throughput

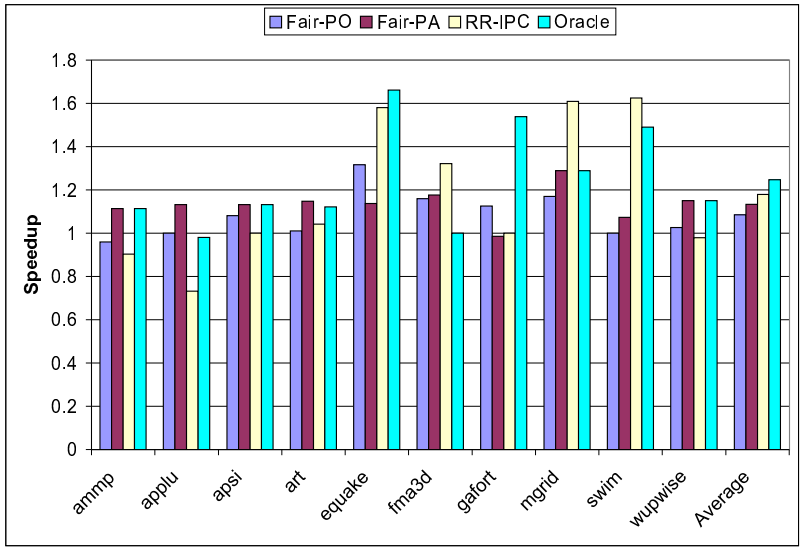


(b) Fair-PA Throughput

Figure 5.1: Thread Throughput for FAIR Schedulers



(a) RR-IPC Throughput



(b) Speedup

Figure 5.2: RR-IPC Thread Throughput and Overall Speedup

5.3.1 Performance

Figures 5.1(a), 5.1(b), and 5.2(a) graph per-thread throughput for the different scheduling methods, normalized to single thread performance on the fastest core, where higher values correspond to better performance. Thread performance on slower cores is accounted for using the equations derived in section 5.1, ensuring that programs running on slower cores are not penalized for substrate heterogeneity. *Max Threads* shows throughput per thread when the application is using all cores. It provides a lower bound on performance. The ideal cases represent throughput when the co-scheduled thread counts are run on the CMP in isolation, for the same phases as the co-scheduled cases. This provides an upper-bound on expected thread throughput when an application is co-scheduled. The difference between the upper and lower bound is the margin for degradation due to contention. Programs that scale well will have very small margins between the one and maximum threads cases, making it difficult for them to be successfully co-scheduled.

Figure 5.1 (a) graphs scheduling performance for the *fair* scheduling when it is processor oblivious (*Fair-PO*), as well as the ideal throughput if there is no contention from co-scheduling (when the programs are run in isolation). The ideal and *Fair-PO* throughput are normalized to the single thread case, accounting for the frequency differences among cores. The *Fair-PO* scheduler can not co-schedule all applications (`applu` and `swim`), and so they are omitted from the throughput graph. Almost all benchmarks have throughput higher than the *max thread* case. The one case where it is slightly lower is `ammp`, where the difference between the one thread and maximum threads case is very small. `art` also has a small margin between the one and maximum threads case. Thread efficiency compared to the maximum threads baseline is reduced by two main factors. One is contention among applications for shared resources, resulting in greater latency

and lower throughput. The second source of inefficiency is bad processor mappings of applications to cores. Assigning threads obliviously between fast and slow cores can result in some memory bound applications running on fast cores that they cannot effectively utilize, or running on slow cores when they would be more energy efficient on faster cores. For example, `swim, applu,` and `equake` are not very efficient on fast cores, which is why two of the three (`swim` and `equake`) fail co-scheduling. `art`, `ammp` and `fma3d` show large performance increases on faster cores, indicating they are not energy efficient with the slow cores. Processor inefficiency affects `ammp` by decreasing energy reductions, which we show in Section 5.3.2 when we examine energy.

Figure 5.1 (b) graphs scheduling performance for the processor aware fair scheduler, denoted as *Fair-PA*. *Ideal S4* and *Ideal F4* show thread performance when running on all the slow processors or all the fast processors respectively. We graph these two combinations since, for our workloads and system configuration, applying the heuristic results in creating all schedules to consist of running applications in solely homogeneous configurations. The *Ideal S4* and *Ideal F4* results show that some benchmarks are just as efficient in thread throughput with four fast threads as they are with four slow threads. Other benchmarks, such as `applu, equake, gafort,` and `swim`, exhibit significant degradation in thread throughput between the *S4* and *F4* policies. The *Fair-PA* results for `wupwise` and `mgrid` show that using the most efficient mappings can make a difference between the *S4* and *F4* performances. For these two benchmarks, a homogenous configuration of slow cores is chosen, and the *Fair-PA* co-scheduled throughput is higher than the *F4* policy, even after contention from co-scheduling. Throughput lower than the baseline does not always result in less performance, since applications can have their time quantum increased disproportionately with the resources they share. While two applications both have their time quanta increased from co-sharing the CMP, the ap-

plication running on the faster cores is losing fewer resources compared to the increase in time quantum (their *fairness*, as defined in equation 5.3, is greater than one). The applications on the slower processors still exhibit a performance increase and benefit from co-scheduling, since the increase in time quantum is of greater benefit than disproportionate core mappings. The high thread throughput of programs running on slow cores hides the performance degradation of badly scaling programs. This phenomenon and the efficiency of mapping applications to cores appropriately is why the *Fair-PA* scheduler is able to successfully co-schedule all the applications within the benchmark suite.

Figure 5.2(a) graphs scheduling performance for the round-robin IPC scheduler (*RR-IPC*). Each application is given one processor on which to run, the mapping is based on its CPU-affinity from Equation 5.1. Results are normalized to the single-threaded case, so the *ideal* value is not graphed like thread throughput graphs 5.1(a) and 5.1(b) since it is the same as the normalization value. Co-scheduling applications together severely degrades thread throughput for several benchmarks, even when applications are mapped to the most efficient cores. For example, `swim` and `wupwise` both execute on slower processors, reducing the thread throughput required for them to match their single thread baseline counterparts. Nevertheless, they still exhibit a 50% reduction in thread throughput from contention for resources. For `wupwise`, the degradation is worse than the *max threads* policy. This is surprising since contention does not significantly effect `wupwise` when paired with programs based on other schedulers (*Fair-PA*, *Fair-PO*). Since the *RR-IPC* scheduler does not check for degradation or ensure *fairness*, performance for several benchmarks degrades from co-scheduling (`ammp, applu,` and `wupwise`). The *RR-IPC* is a simple scheduler that fails to account for the complex underlying substrate on which applications run, resulting in performance loss and energy efficiencies (which is shown in section 5.3.2).

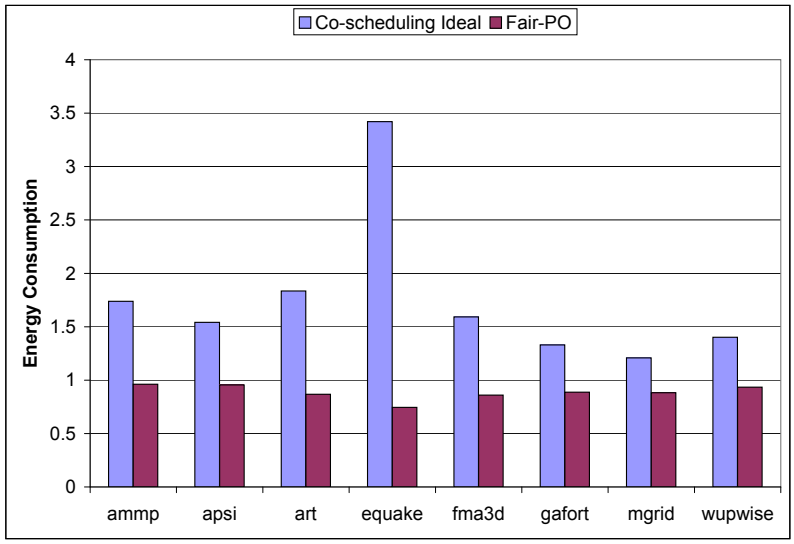
Figure 5.1(b) shows actual performance speedups for the different scheduling methods normalized to the baseline case of maximum thread scheduling. *Oracle* indicates the best scheduling possible assuming oracle knowledge of possible contention among programs. *Average* shows the average speedup of the different scheduling methods across all benchmarks, including benchmarks that are not chosen for co-scheduling. Benchmarks that are not co-scheduled remain at their baseline value, with a speedup of one. The results from *Oracle* scheduling show that there is real potential for improving performance via co-scheduling, with an average speedup of 25%. *RR-IPC* shows the highest average speedups from our three schedulers, however this comes at the cost of degrading performance for two of the eight benchmarks it co-schedules. This violates our condition that no benchmark should be negatively affected from co-scheduling, otherwise programs lose the incentive to share their time quantum with other programs. While `swim` had one of the highest degradations in thread throughput, it shows the highest performance gains with *RR-IPC* scheduling. Resource intensive benchmark `mgrid` shows a large improvement with *RR-IPC* scheduling. This is because these two benchmarks fail to scale at low thread counts, so losing cores is not as critical as increasing their time quantum. *Fair-PA* shows the highest speedup while satisfying our scheduling conditions. Even programs with poor co-scheduled throughput achieve a performance improvement from gaining an unequal time quantum. For example, co-scheduled throughput for `ammp, art, equake,` and `fma3d` are not noticeably better than time-sharing using the maximum number of threads. Performance improvements of 11.3%, 14.8%, 13.7%, and 17.7%, respectively, are achieved. Compute-bound applications are not adversely affected from running on slower cores. This is shown in the results with the *Fair-PO* scheduler. The *Fair-PO* scheduler allocates processors in a frequency oblivious manner, and this improves performance over the baseline for most benchmarks, but not as much as with the *Fair-PA* policy. Only two benchmarks show

better performance with *Fair-PO* instead of *Fair-PA*: `equake` and `gafort`. `equake`'s improvements are a result of application pairing rather than processor mapping, since with the *Fair-PA* scheduler, `equake` is assigned the fastest cores, but the speedup is not as high as with the *Fair-PO* policy. `gafort` however, receives an unfair share of resources, which is why *Fair-PA* has worse performance than *Fair-PO*. These results show that both *Fair-PA* and *Fair-PO* are competitive, and can achieve efficient results under the required scheduling conditions. *RR-IPC* results in contention that can hurt program performance, removing the incentive for co-scheduling some applications.

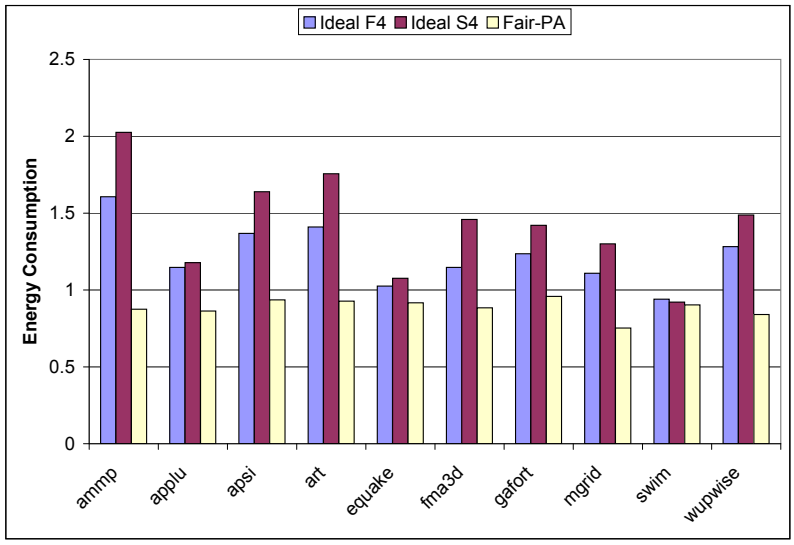
5.3.2 Energy

Figures 5.3(a), 5.3(b), and 5.4(a) graph energy consumption of applications from different co-schedules, where lower is better. All results are normalized to the baseline case of time-shared gang scheduling of each application. The energy reduction of running applications at lower thread counts in isolation is also graphed for reference. If we only consider dynamic power consumption, then the isolated runs would have the least power consumption, since they have the highest throughput of the points graphed. However, due to static power consumption, running applications in isolation results in higher energy consumption, since the overhead of the static power is not amortized over more threads or applications.

Figures 5.3(a) graphs power consumption of the processor oblivious *fair* scheduler, *Fair-PO*. *Co-scheduling ideal* shows energy consumption if the application is not co-scheduled, but run in isolation with the same thread configuration as *Fair-PO*. *Fair-PO* is able to reduce energy consumption of all applications by co-scheduling them, which amortizes the static power cost with secondary applications, unlike the *Co-scheduling*

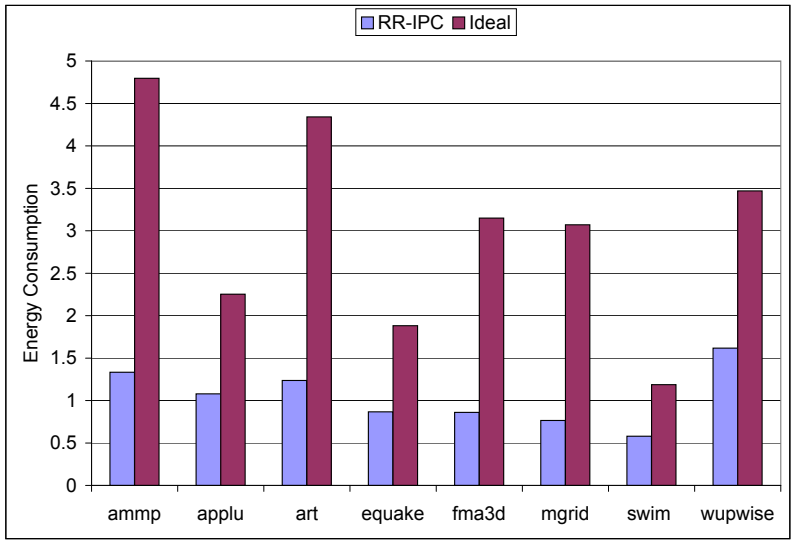


(a) Fair-PO Energy

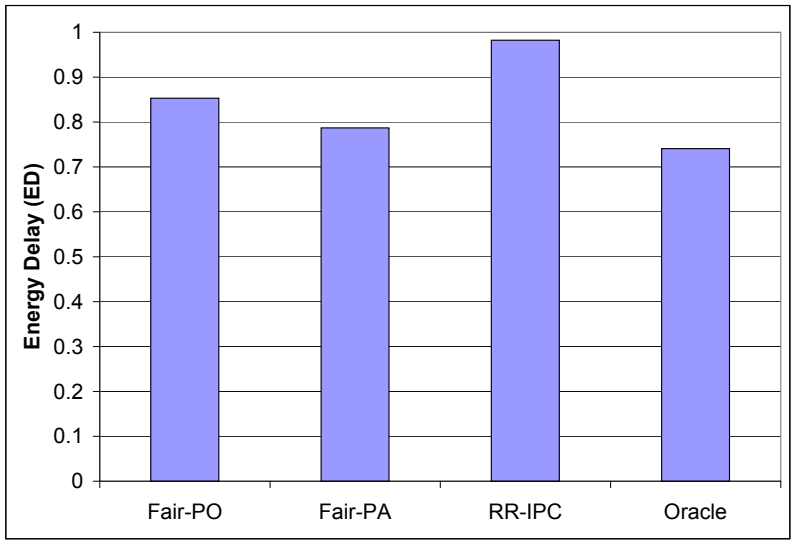


(b) Fair-PA Energy

Figure 5.3: Energy Consumption for FAIR Schedulers



(a) RR-IPC Energy



(b) ED

Figure 5.4: Energy Consumption and Total ED

ideal bars, which shows the high cost incurred by running applications in isolation. The high static power cost of the system favors configurations that lead to faster execution times. Therefore, configurations where applications are mapped to fast and slow cores can be energy inefficient if the applications can benefit from faster processors. Applications like *ammp, apsi, art,* and *fma3d* have a strong affinity to the faster processors, and although these applications do achieve energy reductions, these are meagre for `ammp` and `apsi`, and could have been better (*Fair-PA* results shows this to be the case in Figure 5.3(b)).

Figures 5.3(b) graphs power consumption of the processor aware *fair* scheduler, *Fair-PA*. Energy consumption when running on either all the fast or all the slow cores is also shown (*Ideal F4* and *Ideal S4*). While not all programs can efficiently use the faster cores, using the faster cores almost always results in less energy consumption than using the slower cores. This is because the static power and system power consumption dwarf the dynamic power component of the processor. `swim` is the only benchmark that consumes less energy with *Fair-PA* than using all the cores, even without co-scheduling. This is because `swim` is sufficiently memory bound that increasing thread counts past four fails to improve performance. However, the lowest energy consumption in this case is still the co-scheduled case. Memory bound benchmarks have similar energy consumption between using solely fast or solely slow cores. These applications (`applu, equake, swim`) are well suited for running on slow cores, leaving the faster ones for applications which are the most energy efficient on them. On co-scheduled benchmarks, both *Fair-PA* and *Fair-PO* achieve 12.5% reduction in energy consumption. *Fair-PA* scheduling achieves the best results, since it is co-scheduling 10 applications, unlike *Fair-PO*, which co-schedules only eight.

Our schedulers indicate that co-scheduling always consumes the least amount of en-

ergy, which is not always the case. Energy results for *RR-IPC* co-scheduling are shown in Figure 5.4(a), and demonstrate that contention leads to increased energy costs. These increased energy costs reduce the efficiency of co-scheduling. Only half the benchmarks co-scheduled exhibit reductions in energy consumption. `wupwise` shows increased energy costs, although it has better performance than the baseline. *RR-IPC* actually has the highest ED of all the schedulers, which is surprising since it previously had the highest average performance. The benchmarks that have the best performance degrade the least from co-scheduling. `mgrid` and `swim` again have the highest energy reductions, since contention does not increase their running times or their power overhead.

Figure 5.4(b) graphs average energy times delay (ED) of all the different schedulers, as well as the *Oracle* scheduler. All results are normalized to the baseline case of time shared gang-scheduling at maximum thread counts. *RR-IPC* has the highest ED due to the extra energy consumed from contention among threads. Interestingly, the difference in ED between *Oracle* and *Fair-PA* is less than 5%, while the difference in performance is significantly higher. Although *Oracle* scheduling can achieve significant performance improvements, the energy savings are not as high as our other schedulers. This again indicates that increasing the number of applications co-scheduled past a certain threshold reduces energy savings. While not processor aware, *Fair-PO* remains competitive with other schedulers due to its adaptive nature, and requires fewer time samples to find a solution. However, by being processor aware, ED can be reduced further, by a factor of 1.46, as evidenced by *Fair-PA* scheduling results.

5.4 Conclusions

We create custom multiprogrammed workloads for a frequency heterogenous CMP substrate. These workloads improve thread efficiency by sharing the CMP among programs. Sharing results in programs operating at their most efficient concurrency points. Compared to running programs at maximum thread counts, average application performance is improved by 8.4%. We extend our scheduler by accounting for the heterogeneity of the underlying computing substrate, and we use this information to make better scheduling decisions. This results in performance improvements of 13.4% over the baseline. We compare our work to mapping applications to cores on which they have the highest IPC, and schedule cores round-robin, but we find that this method fails to preserve performance for all benchmarks and achieves the highest energy delay values of all schedulers evaluated. Nevertheless, even this approach is better than the baseline case of time-shared gang-scheduling the CMP for each application, on average. Our processor-aware *fair* scheduler achieves ED efficiency within 5% of the best case *Oracle* value.

With increasing heterogeneity and numbers of cores on chip, appropriately mapping applications to the best cores for their needs and creating efficient schedules will achieve greater levels of application efficiency. This is due to memory and other shared structures not increasing proportionally with the number of cores on chip (resulting in more applications failing to scale), as well as static and off-chip power encompassing a greater portion of the total power of the system (resulting in higher levels of co-scheduling to further amortize static overhead costs).

CHAPTER 6

SUMMARY OF CONTRIBUTIONS

We identify the current problems the community faces (such as the memory bottleneck, and programs that fail to scale with threads). We show why they exist on CMPs, and why they do not manifest themselves on large-scale shared-memory systems. We derive meaningful, hardware-verified scheduling solutions to solve these issues, and demonstrate how these solutions can be implemented in user-space in real-time.

In Chapter 2, we find that off-chip bandwidth limitations can bottleneck applications at low thread counts. Thread contention results in severe backlogs at the memory controller, and increases DRAM page conflicts. These issues are amplified with CMPs since resources external to the core are not duplicated but shared across cores. With multi-socket large shared-memory systems, resources are duplicated, but having separate resources is difficult with CMPs due to physical pin, power, and thermal limitations per socket. Sharing of resources affects scalability, resulting in applications failing to scale at significantly lower thread counts than previously perceived with simulation systems that do not faithfully model main memory. Our work finds that it is insufficient to assume static memory latencies, and that doing so can give erroneous results. We find that accurately modeling main memory is required for any CMP or scalability research.

We demonstrate how to mitigate performance scaling through multiprogrammed co-scheduling, improving performance and energy consumption. We find that using the bandwidth-aware threading metric [56] does not give the optimal thread count for performance or energy, and run-time scheduling and monitoring is required to contend with on-chip contention. We devise heuristics that find efficient solutions quickly at run-time, improve performance by 20%, and reduce energy by a factor of 2.5 over not co-scheduling applications.

We examine scheduling an application's threads for heterogenous cores, assigning different workload sizes to threads depending on the physical core assigned to the thread. We provide methods to determine at run-time which heterogenous configurations are beneficial for performance and energy. We find the OpenMP dynamic scheduling techniques improve over static scheduling for instances of large frequency variation or for compute bound applications.

We combine our Chapter 3 work on co-scheduling and workload balancing for heterogeneous cores to schedule multiprogrammed multithreaded workloads for a heterogeneous substrate. We augment our fair scheduler to account for frequency heterogeneity, mapping applications to cores based on their efficiency. By extending our algorithm, and accounting for the different types of processors within our substrate, we increase the number of applications that can be co-scheduled, and reduce ED by a factor of 1.46 over our processor oblivious co-scheduling heuristic.

As the number of processors on-chip increases, multithreaded programs will be a requirement for achieving increasing performance improvements. Scheduling for a multithreaded landscape is critical to efficiently leveraging current and future CMP hardware. In an era where power is a first-order design constraint, systems on a chip are increasingly becoming the norm, resulting in more programs concurrently sharing resources, which broadens the applicability of this work.

Avenues of future work are to schedule hardware resources instead of software threads. Often, hardware structures are not used all the time, and can be shared when their primary recipient does not require them. For example, caches often have a Non-Uniform Cache Architecture (NUCA) on chip. Proper resource scheduling can effectively multiplex a cache among processors, exhibiting the behavior of a private, large, dynamically sized cache for each processor. As CMPs grow in size, and power contin-

ues to limit what can be embedded within them, managing resources well will be vital for future performance growth.

BIBLIOGRAPHY

[1] AMD Corporation. Model number and feature comparisons - AMD Phenom. Processors. http://www.amd.com/us-en/Processors/ProductInformation/0,,30_118_15331_15332%5E15347,00.html, December 2007.

[2] V. Aslot and R. Eigenmann. Performance characteristics of the SPEC OMP2001 benchmarks. In *Proc. of the European Workshop on OpenMP*, September 2001.

[3] D.H. Bailey, T. Harris, W.C. Saphir, R.F. Van der Wijngaart, A.C. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Report NAS-95-020, NASA Ames Research Center, December 1995.

[4] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The impact of performance asymmetry in emerging multicore architectures. In *Proc. 32nd IEEE/ACM International Symposium on Computer Architecture*, pages 506–517, June 2005.

[5] R. Balasubramonian, D.H. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Dynamic memory hierarchy performance optimization. In *Workshop on Solving the Memory Wall Problem, held at the 27th International Symposium on Computer Architecture*, June 2000.

[6] M. Bhadauria and S.A. McKee. Optimizing thread throughput for multithreaded workloads on memory constrained CMPs. In *Proc. ACM Computing Frontiers Conference*, pages 119–128, May 2008.

[7] M. Bhadauria, V.M. Weaver, and S.A. McKee. Understanding PARSEC performance on contemporary CMPs. In *Proc. IEEE International Symposium on Workload Characterization*, pages 47–56, October 2009.

[8] C. Bienia, S. Kumar, J.P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proc. IEEE/ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, October 2008.

[9] K.A. Bowman, A.R. Alameldeen, S.T. Srinivasan, and C.B. Wilkerson. Impact of die-to-die and within-die parameter variations on the throughput distribution of multi-core processors. In *Proc. IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 50–55, August 2007.

[10] K.A. Bowman and J.D. Meindl. Impact of within-die parameter fluctuations on future maximum clock frequency distributions. In *Proc. IEEE Conference on Custom Integrated Circuits*, pages 229–232, May 2001.

[11] D. Burger, J.R. Goodman, and A. Kägi. Memory bandwidth limitations of future microprocessors. In *Proc. 23rd IEEE/ACM International Symposium on Computer Architecture*, pages 78–89, May 1996.

[12] J.B. Carter, W.C. Hsieh, L.B. Stoller, M.R. Swanson, L. Zhang, E.L. Brunvand, A. Davis, C.-C. Kuo, R. Kuramkote, M.A. Parker, L. Schaelicke, and T. Tateyama. Impulse: Building a smarter memory controller. In *Proc. Fifth Annual Symposium on High Performance Computer Architecture*, pages 70–79, January 1999.

[13] J. Corbalan, X. Martorell, and J. Labarta. Performance-driven processor allocation. In *Proc. 4th USENIX Symposium on Operating System Design and Implementation*, pages 59–73, October 2000.

[14] J. Corbalan, X. Martorell, and J. Labarta. Improving gang scheduling through job performance analysis and malleability. In *Proc. 15th ACM International Conference on Supercomputing*, pages 303–312, June 2001.

[15] V. Cuppu, B. Jacob, B. Davis, and T. Mudge. A performance comparison of contemporary DRAM architectures. In *Proc. 26th IEEE/ACM International Symposium on Computer Architecture*, pages 222–233, May 1999.

[16] M. Curtis-Maury, K. Singh, S.A. McKee, F. Blagojevic, D.S. Nikolopoulos, B.R. de Supinski, and M. Schulz. Identifying energy-efficient concurrency levels using machine learning. In *Proc. 1st International Workshop on Green Computing*, September 2007.

[17] J. Donald and M. Martonosi. Power efficiency for variation-tolerant multicore processors. In *Proc. IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 304–309, October 2006.

[18] Electronic Educational Devices. Watts Up PRO. `http://www.wattsupmeters.com/`, May 2009.

[19] S. Eranian. Perfmon2: a flexible performance monitoring interface for Linux. In *Proc. 2006 Ottawa Linux Symposium*, pages 269–288, July 2006.

[20] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multi-

threaded chip multiprocessors and implications for operating system design. In *Proc. of the USENIX Annual Technical Conference*, pages 26–26, April 2005.

[21] B. Ganesh, A. Jaleel, D. Wang, and B. Jacob. Fully-buffered DIMM memory architectures: Understanding mechanisms, overheads and scaling. In *Proc. 13th IEEE Symposium on High Performance Computer Architecture*, pages 109–120, February 2007.

[22] R. Ge, X. Feng, W. Feng, and K.W. Cameron. CPU MISER: A performance-directed, run-time system for power-aware clusters. In *Proc. International Conference on Parallel Processing*, pages 18–26, September 2007.

[23] S. Herbert and D. Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proc. IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 38–43, August 2007.

[24] E. Humenay, D. Tarjan, and K. Skadron. Impact of process variations on multi-core performance symmetry. In *Proc. ACM/IEEE Design, Automation and Test in Europe Conference and Exposition*, pages 1653–1658, August 2007.

[25] I. Hur and C. Lin. Adaptive history-based memory schedulers. In *Proc. IEEE/ACM 38th Annual International Symposium on Microarchitecture*, pages 343–354, December 2004.

[26] C. Isci, A. Buyuktosunoglu, C.Y. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proc. IEEE/ACM 40th Annual International Symposium on Microarchitecture*, pages 347–358, December 2006.

[27] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Proc. IEEE/ACM 40th Annual International Symposium on Microarchitecture*, pages 359–370, December 2006.

[28] B. Jacob. A case for studying DRAM issues at the system level. *IEEE Micro*, 23(4):44–56, 2003.

[29] A. Jaleel. The effects of aggressive out-of-order mechanisms on the memory subsystem. Ph.D. dissertation, University of Maryland, July 2005.

[30] A. Jaleel and B. Jacob. Using virtual load/store queues (VLSQs) to reduce the

negative effects of reordered memory instructions. In *Proc. 11th IEEE Symposium on High Performance Computer Architecture*, pages 266–277, February 2005.

[31] I. Kadayif, M. Kandemir, N. Vijaykrishnan, M.J. Irwin, and I. Kolcu. Exploiting processor workload heterogeneity for reducing energy consumption in chip multiprocessors. In *Proc. ACM/IEEE Design, Automation and Test in Europe Conference and Exposition*, volume 2, pages 1158–1163, February 2004.

[32] N.S. Kim, T. Austin, D. Baauw, T. Mudge, K. Flautner, J.S. Hu, M.J. Irwin, M. Kandemir, and V. Narayanan. Leakage current: Moore's law meets static power. *IEEE Computer*, 36(12):68–75, December 2003.

[33] T. Kirihata, G. Mueller, B. Ji, G. Frankowsky, J.M. Ross, H. Terletzki, D.G. Netis, O. Weinfurtner, D.R. Hanson, G. Daniel, L.L.-C. Hsu, D.W. Sotraska, A.M. Reith, M.A. Hug, K.P. Guay, M. Selz, P. Poechmueller, H. Hoenigschmid, and M.R. Wordeman. A 390-mm2, 16-bank, 1-Gb DDR SDRAM with hybrid bitline architecture. *IEEE Journal of Solid-State Circuits*, 34(11):1580–1588, 1999.

[34] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded SPARC processor. *IEEE Micro*, 25(2):21–29, March 2005.

[35] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proc. IEEE/ACM 38th Annual International Symposium on Microarchitecture*, pages 64–74, June 2004.

[36] C. Liao, Z. Liu, L. Huang, and B. Chapman. Evaluating OpenMP on chip multithreading platforms. In *Proc. First International Workshop on OpenMP*, June 2005.

[37] W. Lin, S. Reinhardt, and D. Burger. Reducing DRAM latencies with an integrated memory hierarchy design. In *Proc. 7th IEEE Symposium on High Performance Computer Architecture*, pages 301–312, January 2001.

[38] F. Liu and V. Chaudhary. Extending OpenMP for heterogeneous chip multiprocessors. In *Proc. International Conference on Parallel Processing*, pages 161–170, October 2003.

[39] R.L. McGregor, C.D. Antonopoulos, and D.S. Nikolopoulos. Scheduling algorithms for effective thread pairing on hybrid multiprocessors. In *Proc. 19th IEEE/ACM International Parallel and Distributed Processing Symposium*, volume 1, page 28a, 2005.

[40] S.A. McKee. *Maximizing Memory Bandwidth for Streamed Computations*. PhD thesis, School of Engineering and Applied Science, Univ. of Virginia, May 1995.

[41] K. Nesbit, N. Aggarwal, J. Laudon, and J.E. Smith. Fair queuing memory systems. In *Proc. IEEE/ACM 40th Annual International Symposium on Microarchitecture*, pages 208–222, December 2006.

[42] K. Nesbit, J. Laudon, and J.E. Smith. Virtual private caches. In *Proc. 34th IEEE/ACM International Symposium on Computer Architecture*, pages 57–68, June 2007.

[43] H. Oh and S. Ha. A static scheduling heuristic for heterogeneous processors. In *Proc. 2nd International Euro-Par Conference*, pages 573–577, August 1996.

[44] S. Parekh, S. Eggers, and H. Levy. Thread-sensitive scheduling for SMT processors. Technical Report Technical Report, University of Washington, 2000.

[45] José Renau. SESC. `http://sesc.sourceforge.net/index.html`, 2002.

[46] A. Settle, D.A. Connors, E. Gibert, and A. Gonzalez. A dynamically reconfigurable cache for multithreaded processors. *Journal of Embedded Computing: Special Issue on Single-Chip Multi-core Architectures*, 1(1):221–233, December 2005.

[47] C. Severance and R. Enbody. Comparing gang scheduling with dynamic space sharing on symmetric multiprocessors using automatic self-allocating threads (ASAT). In *11th International Parallel Processing Symposium*, pages 288–292, April 1997.

[48] C. Shin, S. Lee, and J. Gaudiot. Dynamic scheduling issues in SMT architectures. In *Proc. 17th IEEE/ACM International Parallel and Distributed Processing Symposium*, pages 77–85, April 2003.

[49] K. Singh, M. Bhadauria, and S.A. McKee. Real time power estimation of multicores via performance counters. *Proc. Workshop on Design, Architecture and Simulation of Chip Multi-Processors*, November 2008.

[50] R.L. Sites. It's the Memory, Stupid! *Microprocessor Report*, 10(10):2–3, August 1996.

[51] A. Snavely and D.M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proc. 9th ACM Symposium on Architectural Support*

for Programming Languages and Operating Systems, pages 234–244, November 2000.

[52] M. Stan and K. Skadron. Guest editors' introduction: Power-aware computing. *IEEE Computer*, 36(12):35–38, December 2003.

[53] Standard Performance Evaluation Corporation. SPEC OMP benchmark suite. `http://www.specbench.org/hpg/omp2001/`, 2001.

[54] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proc. 8th IEEE Symposium on High Performance Computer Architecture*, pages 117–125, February 2002.

[55] G.E. Suh, L. Rudolph, and S. Devadas. Effects of memory performance on parallel job scheduling. *Lecture Notes in Computer Science*, 2221:116, January 2001.

[56] M. A. Suleman, M. K. Qureshi, and Y. N. Patt. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *Proc. 13th ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 277–286, March 2008.

[57] D. Tam, R. Azimi, and M. Stumm. Thread clustering: sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *Proc. of the ACM SIGOPS/EuroSys European Conference on Computer Systems*, pages 47–58, March 2007.

[58] A. Tiwari, S. Sarangi, and J. Torrellas. ReCycle: Pipeline adaptation to tolerate process variation. In *Proc. 34th IEEE/ACM International Symposium on Computer Architecture*, pages 323–334, June 2007.

[59] A. Tiwari and J. Torrellas. An updated evaluation of ReCycle. In *Proc. 7th Workshop on Duplicating, Deconstructing, and Debunking*, June 2008.

[60] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proc. 22nd IEEE/ACM International Symposium on Computer Architecture*, pages 392–403, June 1995.

[61] O. Unsal, J. W. Tschanz, K. Bowman, V. De, X. Vera, A. Gonzalez, and O. Ergin. Impact of parameter variations on circuits and microarchitecture. In *Proc. IEEE/ACM 40th Annual International Symposium on Microarchitecture*, pages 30–39, November 2006.

[62] M. De Vuyst, R. Kumar, and D.M. Tullsen. Exploiting unbalanced thread

scheduling for energy and performance on a CMP of SMT processors. In *Proc. 20th IEEE/ACM International Parallel and Distributed Processing Symposium*, page 10, April 2006.

[63] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob. DRAMsim: A memory-system simulator. *Computer Architecture News*, 33(4):100–107, September 2005.

[64] D. T. Wang. Modern DRAM memory systems: Performance analysis and a high performance, power-constrained DRAM-scheduling algorithm. Ph.D. dissertation, University of Maryland, May 2005.

[65] J. Winter and D.H. Albonesi. Scheduling algorithms for unpredictably heterogeneous CMP architectures. In *Dependable Systems and Networks With FTCS and DCC, DSN*, pages 42–51, June 2008.

[66] P. Wong, H. Jin, and J. Becker. Load balancing multi-zone applications on a heterogeneous cluster with multi-level parallelism. In *Proc. of the Third International Symposium on Parallel and Distributed Computing*, pages 388–393, July 2004.

[67] S. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proc. 22nd IEEE/ACM International Symposium on Computer Architecture*, pages 24–36, June 1995.

[68] Z. Zhang, Z. Zhu, and X. Zhang. Cached DRAM for ILP processor memory access latency reduction. *IEEE Micro*, 21(4):22–32, July/August 2001.