An Approach to Resource-Aware Co-Scheduling for CMPs

Major Bhadauria [*]
Computer Systems Laboratory
Cornell University
Ithaca, NY, USA
major@csl.cornell.edu

Sally A. McKee
Dept. of Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden
mckee@chalmers.se

ABSTRACT

We develop real-time scheduling techniques for improving performance and energy for multiprogrammed workloads that scale non-uniformly with increasing thread counts. Multithreaded programs generally deliver higher throughput than single-threaded programs on chip multiprocessors, but performance gains from increasing threads decrease when there is contention for shared resources. We use analytic metrics to derive local search heuristics for creating efficient multiprogrammed, multithreaded workload schedules. Programs are allocated fewer cores than requested, and scheduled to space-share the CMP to improve global throughput. Our holistic approach attempts to co-schedule programs that complement each other with respect to shared resource consumption. We find application co-scheduling for performance and energy in a resource-aware manner achieves better results than solely targeting total throughput or concurrently co-scheduling all programs. Our schedulers improve overall energy delay (E*D) by a factor of 1.5 over time-multiplexed gang scheduling.

Categories and Subject Descriptors

D.4.1 [**Software**]: Operating Systems—*Process Management*

General Terms

Performance, Algorithms

Keywords

CMP, Scheduling, Performance, Energy Efficiency

1. INTRODUCTION

Processor frequencies no longer increase at historical rates, and thus architects have focused on placing more processor cores on chip to increase parallelism and throughput. This has led to a proliferation of single chip multiprocessors (CMPs) with multiple levels of cache [14]. Multithreaded programs leverage shared-memory architectures by partitioning workloads and distributing them among different virtual threads, which are then allocated to available cores. However, memory speeds still greatly lag behind processor speeds, and observed memory latencies may increase further over uniprocessor chips due to cache coherence checks. This can result in poor performance scaling with increasing thread counts.

Current high-performance scientific computing benchmarks exhibit sub-linear performance gains with increasing numbers of threads due to memory constraints. The standard convention has been to use as many threads as possible, but this may not be the most energy- or performance-efficient solution for multiprogrammed workloads of multithreaded programs. Even if devoting the entire CMP to each application were to deliver best performance, that solution remains infeasible without as many CMPs as applications in the workload. In general, resources must 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 running 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 [3, 24, 6]. In contrast, we discover that for several multithreaded programs better performance results from space-sharing rather than time-sharing the CMP. For cloud computing services, where customers are billed based on compute cycles consumed, running programs in isolation can cost more for cooling and energy than scheduling them together.

We examine memory behavior to derive efficient runtime program schedules for cases where the lack of sufficiently many processors requires that programs be time-shared. 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 present an approach to power-aware thread scheduling for multithreaded programs;
2. We introduce a space-sharing algorithm that holistically allocates processors based on resource consumption;
3. We introduce processor allocation based on the gain and loss of neighboring applications;
4. We infer software behavior by monitoring execution, requiring no hardware modification, user intervention, or recompilation; and

[*]This work was completed while Dr. Bhadauria was a PhD candidate at Cornell University. He is currently a Senior Engineer at EMC Corporation.

5. We verify our energy and performance benefits via real hardware.

In effect, we perform a case study to elucidate scheduling tradeoffs due to workload interactions with respect to resource utilization on CMPs. Our class of co-schedulers (HOLISYN) takes a holistic, synergistic approach to scheduling: we co-schedule applications to achieve better energy and performance than when running them in isolation. We improve total CMP throughput by balancing hardware resource requirements of individual applications. Applications selected to run concurrently complement one another to reduce the probability of resource conflicts. The advantage of our approach is that we can infer software behavior from hardware performance counters, without requiring user knowledge or recompilation. Our methodology asymptotically iterates to find efficient configurations without having to sample the entire solution space of application thread counts and programs that can be co-scheduled. Since we monitor performance at runtime, we revert back to time-sharing the CMP when co-scheduling reduces performance. We examine the power and energy advantages of adaptive co-scheduling and space sharing based on each application's resource consumption. We test our approach on real hardware CMPs, chiefly an eight-core SMP system composed of several dual-core CMPs. We use the PARSEC benchmark suite [4] to illustrate scheduling tradeoffs for emerging multithreaded applications. For poorly scaling programs, we deliver significant performance improvements over both time-sharing and resource-oblivious, space-sharing methods. Our schedulers improve overall performance by 19%, and reduce energy consumption by 26%.

2. RELATED WORK

There has been a large body of work on scheduling for shared resources and quantitatively scaling applications and resources with memory bottlenecks. We briefly cover each of these areas.

2.1 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. Severance and Enbody [20] investigate hybrid time and space sharing of single-threaded and multithreaded codes. They recommend no particular scheduling algorithm, but their results indicate there are performance advantages to dynamic space scheduling single-threaded codes with multithreaded codes. Frachtenberg et al. [11] devise a method of co-scheduling badly scaling programs at run-time. They monitor MPI calls at run-time to determine when a thread is blocked, which unfortunately requires applications to use MPI calls to know when a thread is executing inefficiently. However, an application can be running inefficiently and never block (software synchronization stall), which would not get co-scheduled with their scheduler. Additionally, their implementation of adaptive parallel co-scheduling fails to account for contention between co-scheduling different programs concurrently. Corbalan et al. [6] examine the gains of space-sharing, time-sharing, and time and space-sharing hybrid models. This work evaluates scheduling in the pre-CMP era, with little sharing of resources, resulting in no contention from co-scheduling of applications. The authors 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, decreased, 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 such aspects of the scheduling algorithm design. *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, efficiency is maintained, but enforcing a fair balance of core allocations is not possible. Unfortunately, since interactions among programs are not accounted for, allocations for one program might degrade performance for neighboring applications. *PDPA* also fails to account for power, which has become a critical server design issue. We apply and compare our work to the prior *PDPA* algorithm, highlighting the problem of contention between programs when they are scheduled obliviously.

Corbalan et al. [7] leverage the malleability of programs to change thread counts based on load, reducing context-switching and program fragmentation. McGregor et al. [15] find methods of scheduling programs that complement each other for CMPs composed of SMT processors. Their solutions target workloads with fixed thread counts for each program. Unfortunately, this work requires applications to have static thread counts, which is rarely the assumption of software programmers who design their codes to work for a variety of systems. Our work has no such limitation.

2.2 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 [25]. 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 throughput is not as sensitive to latency as on CMPs: when one thread is bottlenecked, another thread can be swapped into the processor. This is the premise behind the architecture of the Sun Niagara, an eight-core CMP [14]. Parekh et al. [19] predict and exploit resource requirements of individual threads to increase performance on an SMT. De Vuyst et al. [26] look at varying numbers of threads to reduce power consumption of less actively used cores,

Several research efforts [17, 18, 22] 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, attempting to reduce contention for shared resources. Nesbit et al. [17, 18] implement policies that ensure fair resource allocation across applications while trying to minimize performance degradation. Suh et al. [22] promote scheduling program combinations based solely on last-level cache misses. They also propose memory-aware scheduling for multiprogrammed workloads, consisting of single-threaded programs that share the cache [23]. Fedorova et al. [10] estimate the L2 cache miss rates and use their estimates to schedule suitable threads together on a CMP of SMT cores. Their work on mitigating performance degradation is orthogonal to ours. The scheduling, cache, and DRAM channel allocation policies they describe can be leveraged by our scheduler to enforce resource fairness or to reduce contention of co-scheduled programs.

2.3 Thread Scaling

Significant work has been done on scaling of resources when performance counters predict the processor will be resource limited. Isci et al. [13] and Herbert et al. [12] examine scaling frequency when the processor is constrained by memory bottlenecks. Bhadauria and McKee [3] find that memory constraints often render the optimal thread count to be fewer than the total number of processors on a CMP. Curtis-maury et al. [8] predict efficient concurrency levels for parallel regions of multithreaded programs. Suleman et al. [24] examine the most efficient numbers of threads in the presence of bandwidth limitations and data-synchronization. We use their bandwidth-aware threading (BAT) method as our baseline for comparison. None of these studies explores scheduling for several multithreaded programs simultaneously, which we specifically address.

3. RESOURCE-AWARE CO-SCHEDULING APPROACH

Multithreaded programs scale poorly with processor counts for many reasons, including data contention, work partitioning overheads, large serial portions, and contention for shared architectural resources. We focus on mitigating poor scaling from shared-resource contention through co-scheduling. When executing a mix of parallel workloads, time-sliced gang-scheduling improves average job response time, allows processor resource requests to be completely satisfied, provides mutual prefetching among threads sharing memory values via shared caches, and increases sharing of the current working set across threads [16] (reducing contention at all levels of memory). Scheduling and context-switching overheads are generally negligible with respect to the time slice interval. Gang-scheduling all threads of a program ensures that it will not halt at synchronization points (such as barriers and critical sections) waiting for suspended threads. Gang-scheduling CMP resources to each program in equal time quanta enforces fairness. Figure 1 shows how time-sliced gang-scheduling works for three applications on a four-core CMP. Program “App 1” runs at its maximum thread count (here four), with other threads being gang-scheduled on available cores in the first time quantum. At the next timeslice, program “App 2” runs, after which program “App 3” runs, and the process repeats with program “App 1”. This repeats until a program finishes and is removed from scheduling, and remaining programs continue to swap in and out.

Figure 2 is one example of symbiotic time- and space-scheduling, with some applications being co-scheduled, and others being run in isolation. In this hybrid policy, program “App 1” and program “App 2” are each allocated half the number of threads requested. This means each program runs about twice as long as normal, but on half the cores. Program “App 3” 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 quanta (1-3), each program receives an equal amount of all CPU resources. In our hybrid scheduling, fairness is only defined to be true for CPU utilization and not with respect to other resources such as network, file system or memory bandwidth.

For our heuristics to co-schedule programs efficiently, we require applications to satisfy several conditions. These conditions are the need for more cores than are available, sub-linear scaling, malleable thread counts, constant and long application phases (program phases should be long enough for heuristics to converge on a solution within that time), and varied consumption of shared resources. HPC applications designed for large multicore systems often meet these requirements, since they need to ensure the parallel portion of work is greater than the overheads of thread schronization. HPC applications also are designed to work with varying amounts of resources available and generally involve large stable loop structures, such as those seen in the NAS benchmark suite [1].

3.1 Quantifying Shared Resource Activity

If programs share a common resource that cannot be pipelined, such as communication buses 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 statistics provide some insight, but not the complete picture, since programs contend for resources in main memory, at the memory controller, and even at the network interface. Heuristics, by their nature, cannot guarantee that every co-scheduled set gives better performance, since they cannot evaluate all possible schedules.

Suh et al. [23] propose cache miss rates for program scheduling and cache allocation. Last-level cache miss rates indicate what percentage of accesses result in off-chip traffic. Equation 1 defines the last level cache miss rate as the number of misses normalized to the 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 requests are not satisfied by other processors’ 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 requests are still not satisfied.

$$Last\ Level\ Cache\ Miss\ Rate = \frac{Number\ of\ Cache\ Misses}{Number\ of\ Cache\ Accesses} \tag{1}$$

We quantify data bus contention by a vector composed of data bus occupancy and data bus wait times. We order and number applications 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 metrics because all processors share the bus, which is the last shared resource before requests go off chip (we do not have access to performance monitoring counters on the memory controller). Our eight-core system is composed of four dual-core CMPs, where even the two cores on each die use the bus to communicate with each other (snoopy broadcast based coherence protocol). Data bus occupancy represents the time the bus is busy transferring data. Equation 2 defines data bus occupancy as the number of cycles data are transferred on the bus normalized to the total number of program cycles. Suleman et al. [24] use this metric to find the appropriate number of threads for their Bandwidth Aware Threading (BAT) heuristic. Equation 3 shows data bus queue time as data bus wait time normalized to total clock cycles.

$$Data\ Bus\ Occupancy = \frac{Cycles\ Data\ Transferred\ Over\ Bus}{Total\ Cycles\ Program\ Executes} \tag{2}$$

$$Data\ Bus\ Queue\ Time = \frac{Cycles\ Queued\ For\ Bus}{Total\ Cycles\ Program\ Executes} \tag{3}$$

We profile the PARSEC suite to gather average data communication, wait times, and cache miss rates. Table 1 orders applications by increasing bus contention, and Table 2 orders applications by increasing cache miss rates. Table 3 shows the thread counts for lowest theoretical ED for each application.

Cache and bus contention metrics (Table 1 and Table 2) might be expected to give the same orderings, but in practice this is not the case. Some applications, such as `bodytrack`, feature prominently 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 spent 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 absolute time spent transferring data, and are indepen-

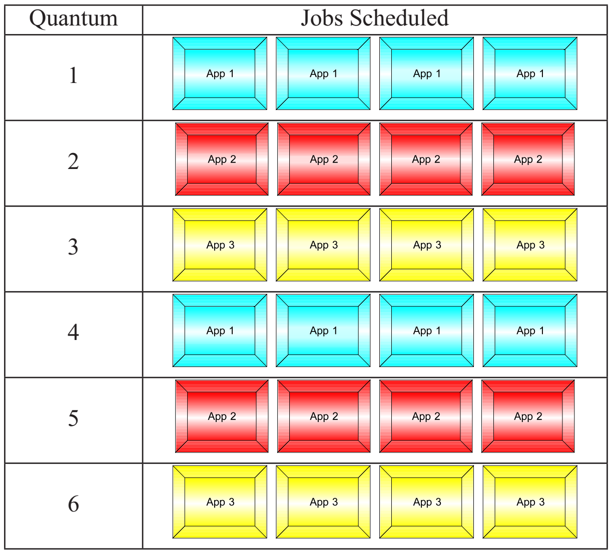


Figure 1: Time-Shared Gang-Scheduling of Three Parallel Applications on a Four Core CMP

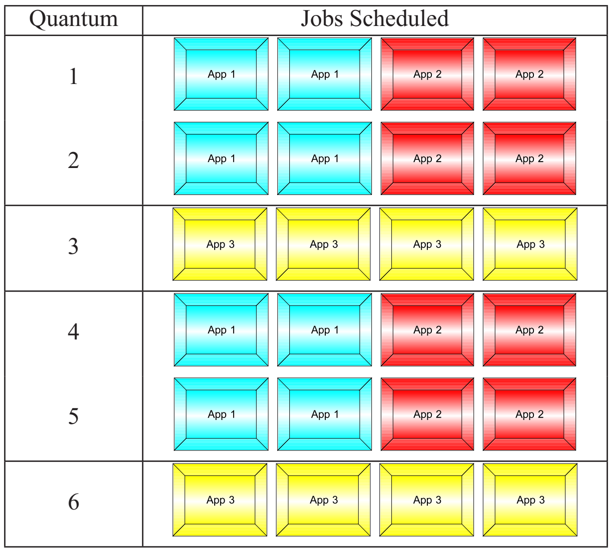


Figure 2: Time- and Space-Shared Gang-Scheduling of Three Parallel Applications on a Four Core CMP

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

Table 1: Applications Ordered by Increasing Bus Contention

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%

Table 2: Applications Ordered by Increasing L2 Cache Miss Rate

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

Table 3: Optimal Thread Concurrency Based on Lowest ED

dent 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, and resulting in data transfers consuming a larger portion of total execution time (in spite of its miss rate). `blackscholes`, a floating-point intensive benchmark, spends more time computing than accessing memory, and therefore its miss rate does not noticeably affect its thread scaling. `canneal` has a 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 different orderings for different metrics.

3.2 Methodology and Metrics for Hardware Aware Scheduling

We present a framework composed of a software performance monitoring unit (PMU) and a performance monitoring scheduler (PMS) that runs as a user-level process. We use this meta-scheduler as a non-invasive proof of concept (as in Banikazemi et al. [2]). Obviously, production systems would incorporate heuristics such as ours into the kernel scheduler (as in Boneti et al. [5]). The PMU examines performance scaling during execution by sampling system performance during an application's time quantum. We feed this information to our scheduler, which reconfigures application mappings as needed.

The PMS supervises executing programs, dictating how many threads they can use. The scheduler incorporates feedback from software/hardware interactions to make future scheduling decisions. Figure 3 shows how 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 the PMU to discern whether throughput for the current application set is sufficiently high, or if rescheduling is needed. The PMU also calls the PMS when it detects an application phase change.

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 (in terms of performance) to the space (processors) it forfeits. We define the conditions for fairness and performance improvement below.

$$Fairness = \frac{New\,Cores}{Old\,Cores} * \frac{New\,Time\,Quantum\,Size}{Old\,Time\,Quantum\,Size} \qquad (4)$$

If a program gives up cores during its quantum, it should be re-

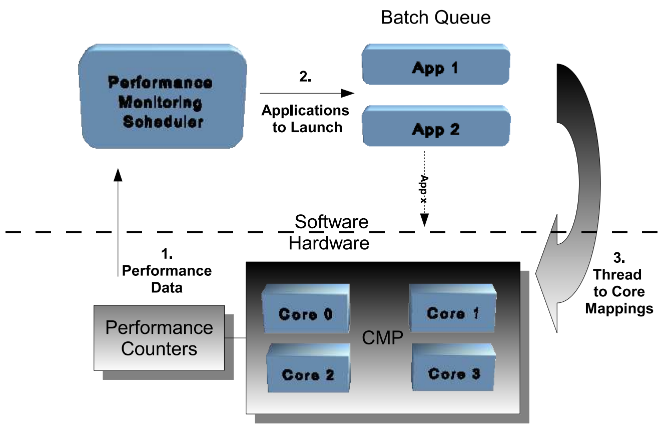


Figure 3: Performance Monitoring Scheduler Overview

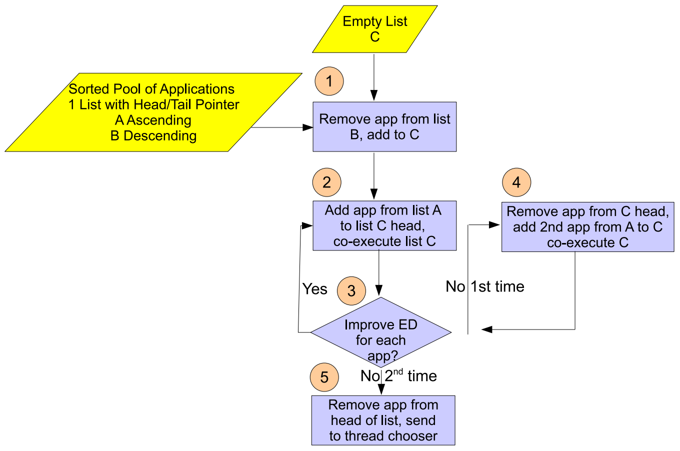

Empty List C
Sorted Pool of Applications
1 List with Head/Tail Pointer
A Ascending
B Descending
1
Remove app from list B, add to C
2
Add app from list A to list C head, co-execute list C
4
Remove app from C head, add 2nd app from A to C co-execute C
Yes
3
No 1st time
Improve ED for each app?
5
No 2nd time
Remove app from head of list, send to thread chooser

Figure 4: Flowchart of Application Chooser Process

warded with extra time equal in clock cycles to space (cores) forfeited. This extra time should come from the time quanta of the application using the forfeited cores. This idea of fairness is quantified in Equation 4, where *fairness* > 1 is more than fair, and *fairness* < 1 is unfair. $NewCores$ is the number an application has after it shares its space, and $OldCores$ is how many were originally scheduled. $OldTimeQuantumSize$ is the original timeslice for each application, and $NewTimeQuantumSize$ is the timeslice once programs are co-scheduled: time quanta are merged, and co-scheduled applications share the same time quantum.

$$NTT = \frac{Instructions\ Retired}{Threads * Execution Time} \tag{5}$$

We define a program's Normalized Thread Throughput (NTT) in Equation 5 as the number of instructions retired normalized by the product of threads and execution time, 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 as the number of threads increases (execution time fails to decrease proportionally). We use this equation to compare thread counts because PARSEC exhibits negligible instruction overhead with increasing numbers of threads. In contrast, when parallelization does increase instruction counts, codes can be instrumented not to count synchronization instructions [8], so instruction overheads of scaling numbers of threads does not distort performance metrics.

Unless an application scales perfectly, it will always increase in thread throughput at lower thread counts. However, co-scheduling may increase contention, which can degrade thread throughput. We quantify thread throughput gain (TTG) (Equation 6) as improvement from using a lower thread count when co-scheduling programs, normalized to throughput at the maximum number of threads for a given co-scheduling candidate. Both the lower thread throughput and maximum thread throughput are computed using Equation 5. Gains of less than one indicate a loss in thread throughput at the lower thread count with co-scheduling. We define local speedup (Equation 7) for an application as the product of *fairness* and TTG. If a program receives less than a fair resource allocation, but the thread throughput gain is very high, it can still experience a speedup. If speedup is greater than one (Equation 7), then co-scheduling improves performance for the application.

$$TTG = \frac{Throughput\ At\ Lower\ Thread\ Count\ With\ Co\text{-}scheduling}{Max\ Thread\ Throughput} \tag{6}$$

$$Speedup = Fairness * (Thread\ Throughput\ Gain) \tag{7}$$

The PMU tracks Performance Monitoring Counters (PMCs) and calls the PMS when a phase change occurs, since thread throughput can change based on the application's current phase. We analyze phase changes by examining events external to the core. We use cache miss rates to indicate actual phase changes, since we are only interested in processor activity that changes the stress put on shared resources. For example, a program may change phases, but if these different phases are not observable outside of the core, they should not change the scaling behavior of that program. We assume our programs are limited by the hardware and not by software design.

4. CO-SCHEDULING POLICIES

We formulate schedulers that choose programs to be run jointly, also choosing the thread counts for each of the programs being co-scheduled. Programs that are not co-scheduled are gang-scheduled. Our goal is to retain the thread throughput that a program exhibits at low thread counts even when it is co-scheduled. For fairness, we assume that each program has equal priority, so our schedules try to improve performance of the overall workload while not overtly penalizing any application (Equation 7 ≥ 1). We explore two heuristics for creating schedules to space-share CMP resources. They take different approaches in choosing applications and concurrency levels. The first strives for fairness, optimizing for both local and global ED. It first chooses applications to co-schedule, and then application concurrency levels. The second optimizes for global ED throughput by co-scheduling jobs at their best local thread counts in a resource-oblivious manner. It first chooses good concurrency levels for each application, and then chooses applications to co-schedule.

4.1 HOLISYN Schedulers

The HOLISYN *fair* (symbiotic) scheduler improves overall *throughput per watt* of the entire workload, while guaranteeing local throughput per watt of individual programs composing that workload. To choose appropriate programs to co-schedule, it samples each to gather communication ratios at the highest thread counts, and then suspends them to main memory. This phase takes place after data is loaded into memory and tasks begin executing in parallel. The approach requires no additional hardware, since current schedulers already require memory and hard drive resources to swap programs when their respective time quanta elapse. Programs that scale almost linearly are removed from the co-scheduling queue and run in isolation since they scale well. We define a term α to indicate how close to ideal scaling is sufficient for a program to not be co-scheduled. There are two main components to our local search heuristic, an application chooser and a thread chooser. The application chooser picks which programs to co-schedule based on resource consumption and co-scheduling interference. The thread chooser picks which concurrency level to use for co-scheduled programs.

After initial application performance sampling, we create a list sorted by resource consumption. The application chooser uses this information to create co-schedules. The doubly linked list is sorted in ascending order with a pointer to the head (list A) as well as a pointer to the tail (for a descending order list B). The application chooser takes the following steps (and is depicted graphically in Figure 4):

1. It chooses the high resource-usage program closest to the top of list B, and adds it to (an initially empty) co-scheduling list C, removing it from the original list.
2. It chooses the low resource-usage program closest to the top of list A, adds it to list C, removing it from the original list. It divides the number of cores round-robin (sorted by lowest resource use) until no more unassigned cores remain. Thus, if N is the number of cores available, each process receives $N/2$ cores.
3. If the EDs for all applications co-scheduled are less than what they were before co-scheduling, then *Step 2* repeats. If the ED for any application is worse after co-scheduling, and if this is the first failed co-scheduling attempt, then the co-scheduler 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 applications in the middle of the list(s), due to their 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 moved to list C. It divides the number of cores round-robin (sorted by lowest resource use) until no more unassigned cores remain. *Step 3* repeats.
5. The program at the head of list C is removed and returned to list A. The programs in list C are sent to the thread chooser, and the application chooser repeats *Step 1* with the set of programs remaining in lists A and C.

The thread chooser takes the following steps:

1. It records the throughput of each program from the application chooser (at equivalent thread counts). Again, for two programs, we measure throughput from equally dividing cores among applications.
2. It increases or decreases thread counts for each program in the list, alternating among applications. This creates two lists, a list containing the reduction in ED of each application from increasing its thread count (list 1), and a list containing the increase in ED of each application from decreasing its thread count (list 2). This requires two time quanta if there are an even number of applications, and three time quanta if an odd number. Generally, a program scales differently, depending on which of its neighboring applications donates the thread it uses for scaling up. We make the simplifying assumption that applications scale independently of other programs with which they are co-scheduled, so the thread chooser does not need to sample $C(\frac{A}{A/2})$ (i.e., A choose $A/2$) different combinations.
3. Numbers of threads are increased for applications in list 1, and decreased for applications in list 2. The process repeats until the applications in list 2 show worse ED than the baseline case of gang-scheduling, or until the overall ED from selectively increasing numbers of application threads decreases. Since the process could potentially continue for N iterations with N cores, we limit the iterations to two to reduce the time quanta required to arrive at a solution.

When increasing thread counts, we must ensure that scaling up an application does not significantly penalize performance of the application being scaled down (i.e., the application whose thread is "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 enforce Equation 4. We therefore extend Equation 5 to derive Inequality 8. This lets us compute delay when comparing equal time intervals in which different thread counts are used for each application.

Inequality 8 specifies the condition that must be satisfied for one application to usurp processor resources, using the previously defined term α to specify how close potential solutions must be to ideal scaling. N is the number of cores before scaling, and $N+1$ is the number of cores after scaling up. Ir_t is the total number of CMP instructions retired before scaling (over some time interval), and Ir_{t+1} is the total number of CMP instructions retired after scaling (over an equal time interval). Inequality 8 serves as an additional condition when populating list 2 in *Step 2* of the thread chooser. This is a secondary constraint, and is not required for the thread chooser stage to function. We limit the value of α via Inequality 9 to ensure that increasing the number of cores does not appear to deliver speedups when there are none.

$$\frac{Ir_{t+1}}{Ir_t*(1-\alpha)} < \frac{N+1}{N} \tag{8}$$

$$\alpha < 1 - \frac{N}{N+1} \tag{9}$$

We experiment with two variations of HOLISYN schedulers that try to ensure fairness: FAIRMIS co-schedules applications by ranking them according to cache miss-rates, whereas FAIRCOM ranks applications according to bus communication.

4.2 Greedy Scheduler

Our next scheduler uses a *greedy* bin-packing heuristic to maximize average system throughput in a resource-oblivious manner. It schedules applications at their most energy-efficient thread counts. Applications are time-sampled at different thread counts, and counts at which they exhibit highest throughput per watt are recorded. Throughput per thread (Equation 5) is divided by total system power consumption, yielding Equation 10. Highest throughput per watt determines the optimal concurrency level for an application. These applications are co-scheduled with others until no unscheduled cores remain.

$$NTT\ /Watt = \frac{Number\ of\ Instructions\ Retired}{Threads * ExecutionTime * Watts} \tag{10}$$

If any cores are left idle, and no applications can fit on the remaining cores, the application with the best scaling properties is allocated more cores so that idle processors are not wasted. Once 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. Programs are scheduled in a resource-oblivious manner, where behavior is assumed to be independent of other applications with which they share the CMP. The drawback of this method is that it requires extensive application profiling in advance (to try multiple numbers of threads and to determine the ED at each thread count). However, most multi-threaded applications lend themselves well to profiling since they often consist of loops, where only a few of the loop iterations need to be run to profile the entire application [8, 24]. This allows the exploration time to be amortized over a longer work span. The *greedy* scheduler takes the following steps:

1. It chooses the best-scaling program (highest concurrency level) from our set of sampled programs, and schedules it on the CMP. If there are no remaining idle cores, it is through scheduling for this program, and repeats *Step 1* for the next program (and time quantum). 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 available idle cores on the CMP, and schedules it concurrently.
3. If there are unscheduled cores remaining, *Step 2* repeats, otherwise co-scheduling is finished for this set. If there are insufficiently many idle cores for any unscheduled application,

then thread counts of the currently scheduled programs are increased. The best scaling program's thread count is increased until performance ceases to improve. The scheduler chooses the best scaling program since it has higher throughput with increasing numbers of threads, and is less likely to overtly consume shared resources.

4.3 Oracle Scheduler

The *oracle* scheduler finds the best case solution when every possible combination of space- and time-sharing applications in the workload is considered, assuming an application cannot be co-scheduled more than once during a round of time quanta. This is the upper-bound on the best performance that can be expected from any scheduler (except cases where applications are co-scheduled within multiple time quanta, i.e., application “A" would be paired with application “B" within one time quantum, and then paired with application “C" within a different time quantum). Note that these results are merely for comparison: the *oracle* scheduler cannot be employed in real systems because no polynomial-time algorithm exists for finding the best co-scheduling solution. Only a brute force search can guarantee the best solution (and verifying the solution takes just as long as finding the solution). In the case of our eight-core CMP with just eight applications, finding optimal schedules required examining over ten thousand samples.

5. 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 [4]. We use these programs to illustrate scheduling benefits, since they represent emerging multithreaded workloads that are of increasing importance in a multicore era. We use publicly available tools and hardware to ensure reproducible 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 (the largest available) to make the workloads as realistic as possible.

We run all benchmarks to completion on an eight core 2.3 GHz dual SMP system with four GB of FB-DIMM main memory. Table 4 indicates relevant hardware parameters. We use the *pfmon* 3.2 utility from the *perfmon2* library to gather data. The Linux *time* command accurately measures execution times of our benchmarks to within tenths of a second. The smallest time granularity used is seconds, since our power meter samples energy consumption per second. We use a Watts Up Pro [9] 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 applications to specific cores, and to set and change the CPU affinity for new or already running applications. Our power model has been hardware-verified to estimate per-core power with very low error [21].

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

Table 4: CMP Machine Configuration Parameters

We use time quanta of two seconds (determined empirically), to ensure each time quantum encompasses a stable period of application behavior. Larger quantum sizes (greater than two seconds) results in longer times until the heuristic converges to a solution (this does not increase the number of reconfigurations performed or change the schedules created). If the sampling quantum for phase detection is very short (i.e. at the microsecond level), a stable phase will never be detected, and reconfiguration will never be attempted. We loop shorter running applications until the longest running application completes (so every application completes at least once). This prevents shorter running programs from skewing results.

6. EVALUATION

We apply our HOLISYN *fair* schedulers to the PARSEC benchmarks, assuming average resource usage based on the data from Tables 1 and 2, from which we derive Tables 5 and 6. The Greedy scheduler uses the thread counts for lowest theoretical ED for each application from Table 3, yielding schedules in Table 7. 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. `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.

We compare the performance of our schedulers with our baseline performance, which consists of running 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 possible;
- this is the optimal number of threads based on their BAT [24] characteristics;
- the programs generally exhibit performance improvements with increasing threads; and
- this is the standard operating procedure in many computing environments.

We next compare against PDPA (Performance-Driven Processor Allocation) [6], which Corbalan et al. [7] find to perform better than several other policies. The PDPA heuristic dynamically adjusts numbers of threads based on scaling at runtime. It uses a resource-oblivious, hill-climbing algorithm to re-evaluate processor-to-thread mappings, and increases or decreases the processors allocated based on the application's meeting a scaling metric. In other words, it examines application performance, making allocation decisions based on the application's scaling efficiency and the 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 (much like our baseline). Applications that do not scale linearly are allocated processors based on scaling efficiency. Note that the original PDPA co-schedules all applications concurrently, and cannot perform time- and space-sharing.

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

Table 5: Workload Configuration with FAIRMIS Scheduling for One Phase

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

Table 6: Workload Configuration with FAIRCOM Scheduling for One Phase

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

Table 7: Workloads and Threads Allocated with Greedy Scheduling for One Phase

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

Table 8: Workloads and Threads Allocated with PDPA Scheduling for One Phase

We modify the algorithm to time-share linearly scaling programs, improving performance.

6.1 Performance

Figure 5 shows thread throughput of co-scheduling strategies normalized to single-thread throughput. Higher thread throughput illustrates improvement in efficiency. Figure 5(a)-(d) graph scheduling throughput for FAIRMIS, FAIRCOM, Greedy, and PDPA. Programs not co-scheduled are omitted.Bars labeled *Co-schedule* illustrate throughput per thread for each application co-scheduled, and those labeled *Co-schedule Ideal* show throughput for each application run in isolation on the CMP using the same thread count as the co-scheduled configuration. This shows what the ideal best case result without contention would look like. *Max Thread* shows throughput per thread when all cores are used. The difference between *Co-schedule* and *Co-schedule Ideal* shows the degradation from resource contention when co-scheduling. With no contention, throughput per thread for co-scheduling is close to *Co-Schedule Ideal*. Differences in throughput between the single-thread and max-thread bars shows the potential speedup that can be achieved by retaining the efficiency of lower thread counts and allocating spare cores to other applications. All results are normalized to the single-thread cases, since they represent the upper-bounds for throughput, unless programs scale super-linearly with increasing threads (which never happens with our benchmarks).

Figure 5(a) graphs thread throughput for FAIRMIS, where cache miss-rates are used to choose programs to co-schedule. Figure 5(a) shows performance degrades minimally for `facesim`, `fluidanimate`, and `canneal` from being co-scheduled with other programs. This is noteworthy for `facesim`, and `canneal` since there is a significant difference in throughput between the ideal and maximum thread cases for those two benchmarks. Other benchmarks degrade noticeably from their ideal values, but still remain a healthy margin above the maximum thread cases.

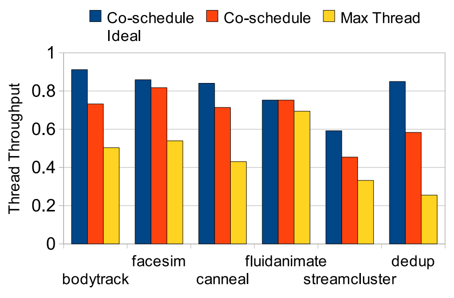
Figure 5(b) graphs thread throughput for FAIRCOM, where co-schedules are chosen based on data bus usage. FAIRCOM shows results similar to FAIRMIS. Performance degrades from the ideal cases, but remains significantly better than for the maximum thread cases. These improvements lead to better performance due to disproportionate time quanta. For example, in Figure 5(b), `fluidanimate` improves by 16% over the baseline, even though thread efficiency is only 3% better; this is because time gains compensate for fewer processors. Similarly, `streamcluster` is 2.09 times more efficient when co-scheduled, but only exhibits a speedup of 1.54, because time gains do not compensate for forfeited processors.

Figure 5(c) graphs thread throughput for the Greedy scheduler. `facesim` degradation from contention is worse than the maximum thread case, and even if it receives a fair allocation of time for resources given, its performance is also worse than the baseline. Table 7 shows that when scheduled greedily, its time quantum increases by a factor of three, but its loss in processors is less than a factor of three; in this case performance improves by 37% over the baseline. 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 optimal. Other programs remain a healthy margin above the maximum thread case, keeping Greedy competitive with other scheduling methods.

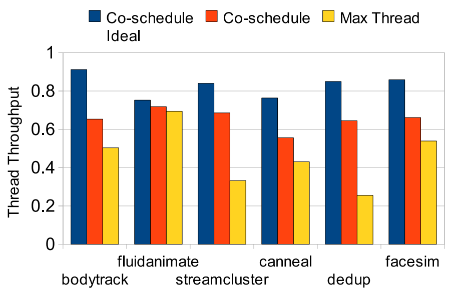
Figure 5(d) graphs thread throughput for the PDPA scheduler. All benchmarks suffer some performance degradation from coscheduling. `bodytrack` suffers severe contention, with thread throughput dropping to the level of the maximum thread case. `facesim` and `streamcluster` are only allocated one processor. `facesim` loses performance when co-scheduled, since its efficiency over the maximum thread case does not compensate for the unfair resource distribution it receives for sharing its time quantum. The PDPA algorithm cannot account for fairness in the presence of thread contention, thus it can generate inefficient or unfair schedules.

Figure 6 shows overall speedups of the different schedulers normalized to the baseline case of time-shared gang-scheduling. Bars labeled *Scheduled* show speedups for applications chosen for co-scheduling, and those labeled *Overall Average* show speedups for the entire suite, including programs not chosen for co-scheduling. The number of programs chosen for co-scheduling is shown next to each scheduler. The more programs co-scheduled, the smaller the difference is between *Scheduled* and *Overall Average*. Co-scheduling delivers results close to Oracle scheduling. FAIRCOM performs as well as the Oracle, and FAIRMIS is very competitive. Greedy and PDPA show lower gains due to higher contention among co-scheduled programs. While the differences in speedup across co-schedulers are not significant, the performances of individual applications vary greatly, since most of the schedulers do not guarantee fairness for co-scheduled applications. FAIRCOM and FAIRMIS co-schedule more programs, since Greedy is more conservative about choosing applications to schedule together. The PDPA equal-partition variant tries to co-schedule as many applications together as possible, resulting in too many co-scheduled programs, leading to the highest contention levels.

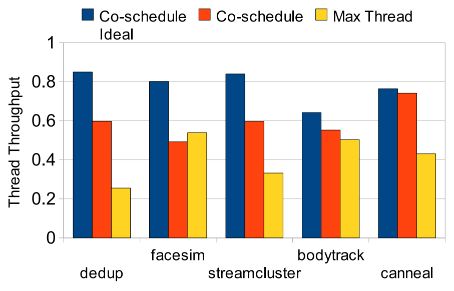
The HOLISYN *fair* algorithm underlying FAIRCOM and FAIRMIS has several benefits. First, it is simpler than the Greedy scheduler, because the application's optimal concurrency level is not required, therefore we need not profile applications at all concurrency levels. Second, using the *fair* algorithm requires only the performance, cache, and off-chip characteristics to be calculated for one thread count (the largest). Additionally, we find that the *fair* algorithm can improve performance for more programs. For example, while Greedy, FAIRCOM, and FAIRMIS achieve similar performance improvements, the *fair* schedulers co-schedule more pro-



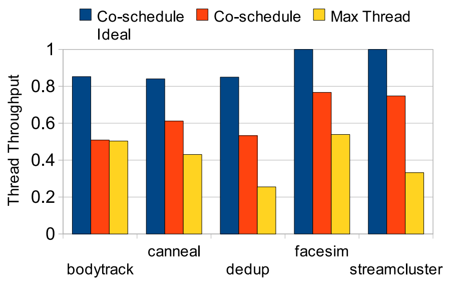
(a) FAIRMIS



(b) FAIRCOM



(c) Greedy



(d) PDPA

Figure 5: Thread Throughput Normalized to Single-Thread Configuration (Higher is Better)

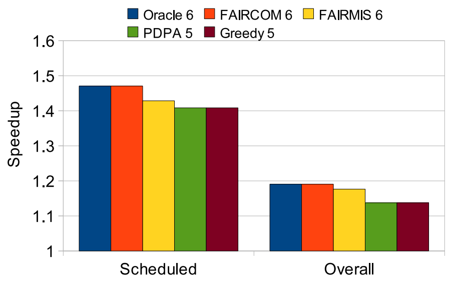


Figure 6: Speedup Normalized to Standalone Execution (Higher is Better)

grams, leading to higher overall performance.

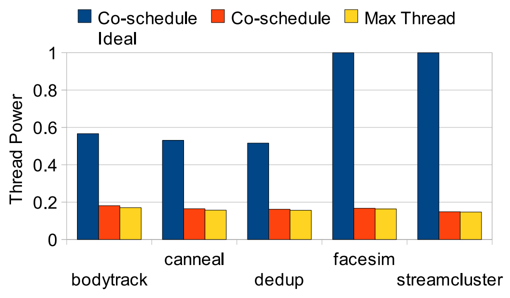
FAIRCOM performs better for a few more benchmarks than FAIRMIS. Monitoring communication latencies provides more insight for creating co-schedules than tracking cache misses: the bus contention vector is a better indicator of sharing contention, since its values are normalized to program run times, while memory misses are normalized to cache misses (which fails to fully convey an application's dependence on memory for performance). For example, the three programs that scale almost linearly also exhibit the least bus contention. Both HOLISYN schedulers demonstrate that co-scheduling helps programs retain their low thread-count ED efficiency on larger, multicore CMPs by efficiently leveraging idle resources for other tasks.

6.2 Power and Energy

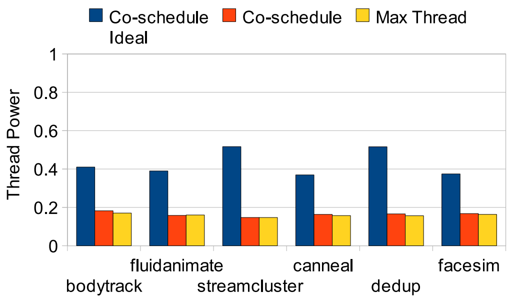
We examine the total system power consumed by each thread of the different workload configurations. While single-threaded benchmarks have the highest throughput, running them results in total system power being amortized over only a single thread. Total system power does not scale linearly with increasing numbers of active cores on the CMP, since static power of off-chip components (memory, interconnect) and on-chip memories is consumed 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 the number of threads increases, the static power of components is amortized over a greater number of threads.

Figure 7 graphs power consumption per thread for applications executed at different thread counts for the PDPA and FAIRCOM. We omit the FAIRMIS and Greedy graphs since they show similar trends. Power per thread is normalized to the single-thread configuration (lower values are better). Power per thread decreases as number of threads increases, with the co-scheduled workloads or the workloads at maximum thread counts exhibiting lowest costs. At lower thread counts, the benchmarks actually use almost the same or less power as when executing in isolation. This is interesting, because at lower thread counts, the applications might be expected to consume more dynamic power because they commit more useful instructions per time quantum. However, if static/system ("uncore") power accounts for most of the total power, then total power changes little when the system runs at more efficient concurrency levels. Additionally, dynamic power consumption may decrease if thread contention is reduced. Unlike the maximum thread case, benchmarks are more power efficient (in terms of performance/watt) at lower concurrency levels, and extra cores can also run other benchmarks at power-efficient concurrency levels. The graph shows that watts per thread does not change from the maximum thread cases. This is interesting because more work is performed (as shown in earlier graphs with higher thread throughput). Co-scheduling significantly reduces power per thread over the co-scheduling ideal case, since benchmarks are able to share system (uncore) power consumption overhead with other programs.

We calculate energy using total application power consumption (from our per-core power derivations) and elapsed time, since energy is the product of power over time. Figure 8 shows average energy reductions normalized to the baseline of time-shared gang-scheduling. Bars labeled *Scheduled* represent average energy reduction for co-scheduled programs, and those labeled *Overall Average* show average reduction for the entire benchmark suite. Interestingly, programs with worse performance than the baseline may still see energy savings (since the performance degradation results



(a) PDPA



(b) FAIRCOM

Figure 7: Total Power Consumption per Thread Normalized to Single-Thread Configuration (Lower is Better)

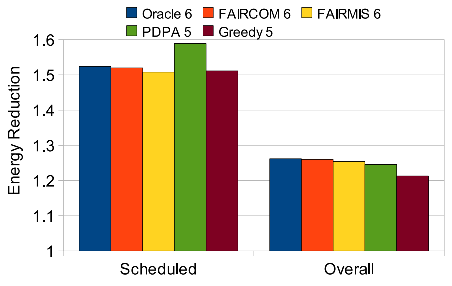


Figure 8: Energy Reduction Normalized to Gang-Scheduling (Higher is Better)

from resource unfairness), and programs with better performance might see higher energy expenditures due to increased contention (e.g., increased DRAM activity). Benchmark energy reductions are difficult to predict, since threads react differently to increases in throughput. Although individual benchmark performance is not graphed, benchmarks such as `facesim` and `canneal`, exhibit proportional reductions in energy consumption with performance improvements. Other benchmarks, such as `bodytrack`, show only minor improvements in energy consumption compared to throughput improvements. For example, when `facesim` is co-scheduled using PDPA, performance degrades by 11%, but energy consumption decreases by 1.39. PDPA scheduling shows the highest energy reductions for co-scheduled programs, but at the expense of other applications, with programs such as `bodytrack` experiencing increased energy usage. FAIRCOM scheduling shows the second highest energy reductions for co-scheduled programs, and the highest reductions for the entire suite, since it reduces energy for all programs co-scheduled, and chooses the most programs for co-scheduling. Most noteworthy is that requiring all co-scheduled programs to benefit from co-scheduling does not hinder the performance of our scheduling heuristics: HOLISYN schedulers deliver the best performance compared to other heuristics that optimize only for global throughput.

7. CONCLUSIONS

We investigate multithreaded programs that are constrained by shared resources, and increase their collective throughput via holistic, symbiotic (HOLISYN) co-scheduling (which implements both time- and space-sharing). Using an eight-core CMP, we first devise metrics to assess program resource requirements, and then use this information to better schedule both single-threaded and multithreaded workloads. Instead of scaling frequency for bandwidth-limited CMPs, we reduce the numbers of threads and schedule programs together to improve individual benchmark efficiency and overall workload performance. On the PARSEC suite, our HOLISYN schedulers perform better than previous co-schedulers, minimizing contention by co-scheduling programs that complement each other with respect to use of shared resources. Our methodology works for cases when applications fail to scale by design, as well as when they are limited by the available hardware resources. We profile performance and energy at run time, which ensures that our schedules can adapt to changes in program behavior.

We show that Bandwidth Aware Threading (BAT) by Suleman et al. [24] does not provide the optimal thread count for performance or energy, and thus run-time scheduling and monitoring is required to manage 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 schedulers 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 code, since we derive our scheduling information from hardware performance counters. 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. For poorly scaling programs, we deliver significant performance improvements over both time-sharing and resource-oblivious, space-sharing methods. Our schedulers improve overall performance by 19%, and reduce energy consumption by 26%. Through co-scheduling, we reduce ED by a factor of 1.5 over time-multiplexed gang scheduling.

Future work will examine scheduling for heterogenous cores within a CMP and dynamically partitioning the workloads during program execution via run-time 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 CMPS constrained by shared resources.

8. REFERENCES

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

[2] M. Banikazemi, D. Poff, and B. Abali. PAM: A novel performance/power aware meta-scheduler for multi-core

systems. In *Proc. IEEE/ACM Supercomputing International Conference on High Performance Computing, Networking, Storage and Analysis*, number 39, Nov. 2008.

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

[4] C. Bienia, S. Kumar, J. 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, Oct. 2008.

[5] C. Boneti, R. Gioiosa, F. Cazorla, and M. Valero. A dynamic scheduler for balancing HPC applications. In *Proc. IEEE/ACM Supercomputing International Conference on High Performance Computing, Networking, Storage and Analysis*, number 41, Nov. 2008.

[6] 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, Oct. 2000.

[7] 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.

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

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

[10] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *Proc. USENIX Annual Technical Conference*, pages 26–26, Apr. 2005.

[11] E. Frachtenberg, D. G. Feitelson, F. Petrini, and J. Fernandez. Adaptive parallel job scheduling with flexible coscheduling. *IEEE Transactions on Parallel and Distributed Systems*, 16(11):1066–1077, Nov. 2005.

[12] 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, Aug. 2007.

[13] 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, Dec. 2006.

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

[15] R. McGregor, C. Antonopoulos, and D. 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, Los Alamitos, CA, USA, Apr. 2005. IEEE Computer Society.

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

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

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

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

[20] 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, Apr. 1997.

[21] K. Singh, M. Bhadauria, and S. McKee. Real time power estimation of multi-cores via performance counters. *Proc. Workshop on Design, Architecture and Simulation of Chip Multi-Processors*, Nov. 2008.

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

[23] 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, Feb. 2002.

[24] 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, Mar. 2008.

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

[26] M. D. Vuyst, R. Kumar, and D. 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, Apr. 2006.