Scalability-Based Manycore Partitioning

Hiroshi Sasaki
Kyushu University
744 Motooka Nishi-ku
Fukuoka, Japan
sasaki@soc.ait.kyushu-u.ac.jp

Teruo Tanimoto*
The University of Tokyo
7-3-1 Bunkyo-ku, Hongo
Tokyo, Japan
tanimoto@hal.ipc.i.u-tokyo.ac.jp

Koji Inoue
Kyushu University
744 Motooka Nishi-ku
Fukuoka, Japan
inoue@ait.kyushu-u.ac.jp

Hiroshi Nakamura
The University of Tokyo
7-3-1 Bunkyo-ku, Hongo
Tokyo, Japan
nakamura@hal.ipc.i.u-tokyo.ac.jp

ABSTRACT

Multicore processors have been popular for years, and the industry is gradually shifting towards the era of manycore processors. Single-thread performance of microprocessors is not growing at a historical rate, but the existence of a number of active processes in the computer system and the continuing development of multi-threaded applications benefit from the growing core counts to sustain system throughput. This trend brings us a situation where a number of parallel applications simultaneously being executed on a single system. Since multi-threaded applications try to maximize its throughput by utilizing the whole system, each of them usually create equal or larger number of threads compared to underlying logical core counts. This introduces much greater number of threads to be co-scheduled in the entire system. However, each program has different characteristics (or scalability) and contends for shared resources, which are the CPU cores and memory hierarchies, with each other. Therefore, it is clear that OS thread scheduling will play a major role in achieving high system performance under such conditions. We develop a sophisticated scheduler that (1) dynamically predicts the scalability of programs via the use of hardware performance monitoring units, (2) decides the optimal number of cores to be allocated for each program, and (3) allocates the cores to programs while maximizing the system utilization to achieve fair and maximum performance. The evaluation results on a 48-core AMD Opteron system show improvements over the Linux scheduler for a variety of multiprogramming workloads.

*The author is currently with FUJITSU LABORATORIES LTD.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management—*scheduling, multiprocessing/multiprogramming/multitasking*

General Terms

Experimentation, Measurement, Performance

Keywords

Multicore/manycore processors, process scheduling, multi-threaded applications, scalability, performance prediction, dynamic optimization

1. INTRODUCTION

The end of clock frequency scaling has led the processor architecture design towards multiple processors on a chip (known as *multicore processors* or *CMPs*) which exploits thread level parallelism to improve performance [14]. The trend of increasing the core count tends to continue and push the industry and researchers to face *manycore processors* in the near future [24, 27]. The shift from single-core to multicore has broaden the range of applications, and the focus of research has shifted from executing one single-threaded application on a system to either (1) concurrently executing multiple single-threaded applications or (2) running one multi-threaded application on a system. However, considering that not all programs can best utilize the potential of manycore architectures, further shift towards manycore processors will force us to manage dozens of simultaneously executed programs which are heavily multi-threaded.

In such a scenario, shared resource contention is one of the most fundamental and important problems which have to be solved. Prior work have showed that contention should be minimized by (1) isolating (or partitioning) shared resource accesses as much as possible, and at the same time, (2) allocating appropriate amount of shared resources to each program [23]. For single-threaded multiprogramming workloads, trying to reduce contention at the last level cache (LLC), memory controller or the prefetcher which lie down in the memory hierarchy is the main objective [2, 5, 6, 12, 15, 19, 30]. However, memory hierarchy is not the only shared resource when scheduling multi-threaded applications. Parallel programs tend to be executed by creating equal or larger number of threads than the underlying logical core counts to fully make use of the system. Therefore, the computing resources (i.e., CPU cores) which directly contribute to performance is the most important shared resource that we have to carefully allocate to each program in order to achieve high performance.

Allocation should be done dynamically according to program's characteristics at the point of creation or termination of programs, and therefore the runtime layer such as the OS scheduler or the programming language runtime system [16, 21] will play a major role in it. Traditional OS scheduler tries to assign equal CPU time to each thread to ensure the *fairness* property. Therefore, we can consider that CPU cores are evenly shared among each program, controlled by the OS scheduler. This approach is simple and fair from an OS point of view, however, there is still room for improvement and we can still perform better in terms of performance by taking the *scalability* of each program into account.

In this paper, we propose a sophisticated OS scheduler for multi-threaded multiprogramming workloads on manycore processors. The proposed scheduler is able to optimize against various performance metrics which can be determined by either the user or the system software. Several performance metrics are used to evaluate the multiprogramming workloads in the literature: total throughput, weighted speedup [28], harmonic mean of the per program speedup [18], or average normalized turnaround time (i.e., execution time) or ANTT [11]. We insist that the manycore OS scheduler should be flexible enough to optimize against different needs (or metrics) and the key to achieve this is to take the scalability of each program into account. The principle behind our technique is that the speedup obtained from multi-threading on manycore processors heavily depends on several details of the program, and are quite different between them. For example, some program might double its performance by doubling the number of cores, while other might achieve no performance improvement or even degrade its performance. Therefore, we can achieve better performance than the evenly (or fairly) partitioned allocation which is achieved in the Linux scheduler by allocating optimal number of cores to each program according to their scalability.

The main contributions of this paper include the following:

- We find that state-of-the-art Linux default scheduler performs poorly in a multi-threaded multiprogramming environment because of lacking the consideration of scalability of its scheduled programs.
- We propose a scheduling algorithm which predicts the scalability of multi-threaded applications to dynamically allocate the appropriate number of cores to each program in order to optimize system performance. We also introduce a couple of sophisticated enhancements to the scheduler which attempt to get the most out of manycore environments.
- We implement a practical user level scheduler called SBMP (pronounced *S-Bump*) scheduler which (1) requires no changes to the OS kernel, (2) works on existing Linux systems and uses existing hardware performance monitoring units (PMUs), and (3) works with any unmodified application binaries.
- We evaluate the proposed SBMP scheduler with an AMD 48-core system, which shows our design well outperforms the Linux default scheduler for a series of multiprogramming workloads comprised of emerging PARSEC benchmark suite [4].

The remainder of the paper is structured as follows. Next section motivates our work on developing *scalability-based manycore partitioning scheduler (SBMP scheduler)* by showing the poor performance of Linux default scheduler when multiple multi-threaded workloads are simultaneously executed on a manycore environment. Section 3 describes our scheduler algorithm and implementation in detail along with some novel improvements added to enhance our scheduler design. Section 4 explains our evaluation framework and Section 5 shows the performance of the proposed scheduler with wide variety of evaluations. Section 6 introduces related work, and finally the conclusions and future work are stated in Section 7.

2. MOTIVATION

Figure 1 shows how Linux default scheduler performs poorly in a multi-threaded multiprogramming environment. The figure shows performance result of four multi-threaded applications (`canneal`, `ferret`, `raytrace` and `streamcluster` chosen from PARSEC benchmark suite) when they are simultaneously executed on a 48-core symmetric multi-processor system[*]. Bars in Figure 1 indicate the normalized turnaround time (NTT) [11]. NTT is a lower-is-better metric which has a value larger than or equal to one[†], and quantifies the turnaround time slowdown due to multiprogrammed execution. Right most group of bars shows the average NTT (ANTT). ANTT is a reciprocal of the commonly used harmonic mean metric of the relative throughputs under multiprogramming environment compared to their solo-runs [18]. ANTT has a system-level meaning rather than the harmonic mean metric, and we believe it is a good indicator to represent system performance in multiprogramming environments. Therefore, we use this metric throughout this work. The results of Linux default scheduler and the *best static partitioning* are shown in the figure.

For these applications, the best partitioning (or core allocation) is: 12 cores for `canneal` and `raytrace` each, 18 cores for `ferret`, and six cores for `streamcluster`. Note that all programs create 48 threads (or more, according to the program) in order to fully utilize the processor, and threads of each program are packed onto the specified number of cores [7] when partitioned. The best partitioning was obtained by trying all the possible allocations. There are several ways to assign the cores to programs even the number of cores to be assigned are the same. Figure 3 shows two possible assignments which are (a) distributing or (b) centralizing the threads on the system. We have compared the ANTT for these two types of assignments for all the workloads and found that centralizing the threads gives higher or

[*]Figure 2 shows the base node architecture of the evaluation system we use throughout this paper. It consists of four CMPs with each CMP having 12 core multi-chip module processor consisting of two six core dies. Further details of the experimental environment is described in Section 4.

[†]Note that the origin of all the graphs which show the NTT begins with one.

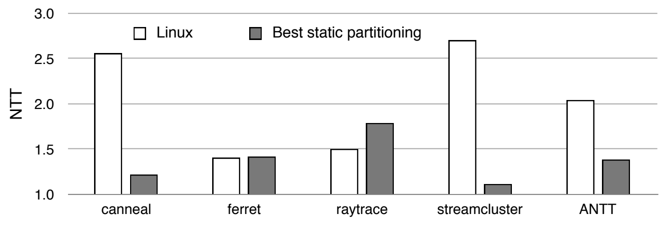

Figure 1: Normalized turnaround time (NTT) of Linux scheduler and static space sharing.

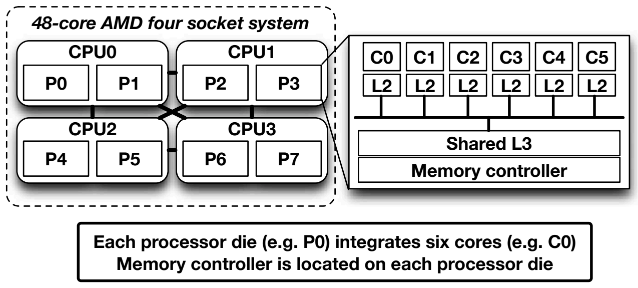
48-core AMD four socket system
CPU0
CPU1
CPU2
CPU3
P0
P1
P2
P3
P4
P5
P6
P7
C0
C1
C2
C3
C4
C5
L2
Shared L3
Memory controller
Each processor die (e.g. P0) integrates six cores (e.g. C0)
Memory controller is located on each processor die

Figure 2: Node architecture of the evaluation platform.

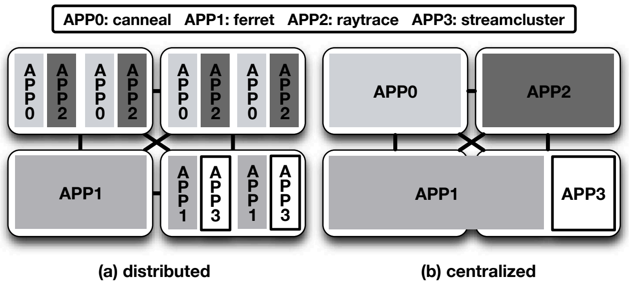
APP0: canneal APP1: ferret APP2: raytrace APP3: streamcluster
APP0
APP2
APP1
APP3
(a) distributed
(b) centralized

Figure 3: Two possible program-to-core assignments of best partitioning.

at least the same performance in most cases. This is quite intuitive because of the low latency of accessing the shared data (including coherency messages) and minimized interference or contention between programs. Therefore, we use the centralized assignment of cores throughout this work.

Also, we decided to use the processor die in Figure 2 as a minimum unit of allocation to programs. That is, each program is provided a number of cores which is a multiple of six in the experimental platform. This restriction still leaves us plenty of room for optimization, while eliminating the unexpected contention by potentially giving exclusive use of the portion of LLC which is on the same die. We can say that the memory subsystem is partitioned as well as the cores, which means that the program with a greater number of cores also gets larger amount of LLC. More sophisticated memory managements such as applying page coloring technique [17] in order to give larger amount of LLC to a program with fewer number of cores can be considered, but those optimizations are left for future work, and we focus on the scheduling of CPU cores in this study. Note that we dedicate each core to only a single program at a time, which means that no two programs share the same core[‡]. These partitioning methods are also used in the proposed scheduler.

We can see the poor ANTT of the Linux default scheduler from Figure 1 which is 2.04, where the ANTT of best static core allocation is 1.38, and significantly improves over the Linux scheduler. To understand the reason of the result shown in Figure 1, we display the scalability of the PARSEC applications in Figure 4. The x-axis indicates the number of cores allocated and y-axis shows relative performance to the maximum. The benchmarks are classified into three groups named with colors according to their scalability characteristics, namely *Green*, *Yellow* and *Red*, which are used in the later section for discussions. The numbers on top of each graph show the maximum speedup obtained compared to the case when a single core is assigned.

The key factor of this improvement is to appropriately controlling the number of cores to be assigned to each application by considering their scalability. This can be seen from the scalability curve of the evaluated four programs, especially `ferret` and `streamcluster`. The best partitioning allocates 18 cores to `ferret` and six cores to `streamcluster`. `Streamcluster` achieves its maximum performance at six cores so there is no reason to allocate greater number of cores from a performance point of view. Other three programs show similar shapes of scalability curve, however, the number on top of the graph showing the maximum speedup against a single core assignment is the largest for `ferret` which is 17.22x. Therefore, it is beneficial to give the resource to `ferret` to maximize the multiprogramming performance. This result motivates us that the scalability of programs should be taken into account by the OS scheduler to achieve good performance.

We have seen the potential performance improvement of static partitioning approach thus far, however, it is not practical to apply the static scheduling in a general purpose OS scheduler because: (1) the best core assignment cannot be easily obtained in advance without the knowledge of the scalability of programs *a priori*, (2) the best core allocation might change during runtime because applications can be composed of several phases which shows different scalability characteristics, and (3) applications are dynamically executed on a general purpose OS so that we do not know which applications will be co-scheduled at what time. We address the above issues by dynamically detecting the program's *scalability* along with several enhancements which are stated in Section 3.

3. SCHEDULER DESIGN

We designed and implemented **SBMP** scheduler, a *scalability-based manycore partitioning* scheduler which achieves high system performance under multi-threaded multiprogramming execution. SBMP scheduler dynamically predicts the scalability of each multi-threaded application by using a simple performance model and select the appropriate amount of cores to be assigned. The model is obtained by slightly modifying the Amdahl's law equation. The key idea behind this allocation is that, we do not give fair amount of CPU time and resources as in the traditional OS scheduler, instead we provide the resources considering their scalability,

[‡]Later in Section 3, we propose an enhancement to the scheduler which relaxes this constraint to allow up to two programs sharing each core to improve performance.

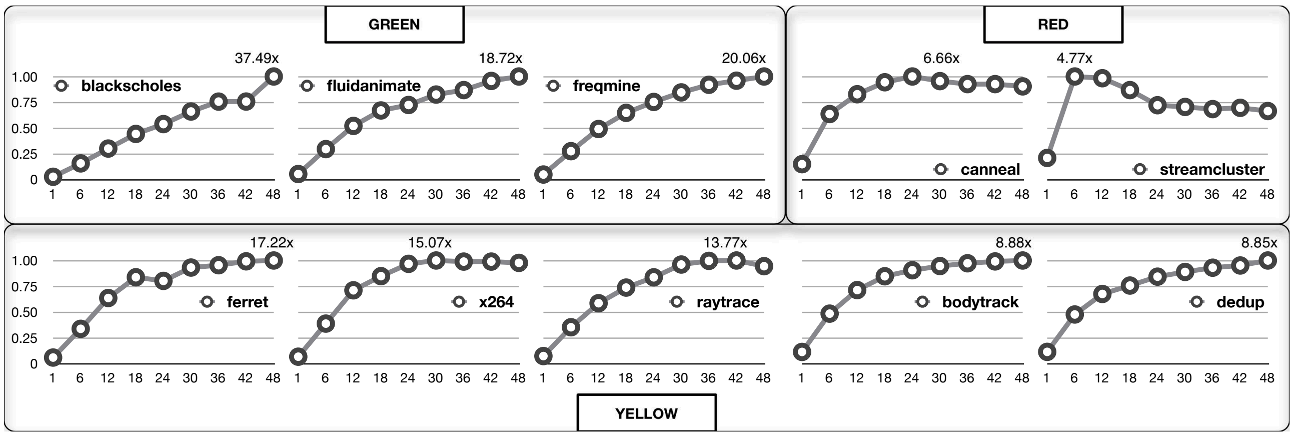

Figure 4: Scalability of the evaluated PARSEC benchmarks.

giving more resources to programs that make more use of them. That is, SBMP scheduler attempts to balance the utilization of resources, not the physical amount of resources. This section provides the scheduling overview, as well as the description of the prediction model and its usage, and the scheduling algorithm and implementation of SBMP scheduler in detail.

3.1 Scheduler Overview

The basic overview of the scheduling algorithm is shown in Figure 5. The top view of the algorithm is fairly simple constructing a loop which consists of the following: (1) SBMP scheduler decides to change the core partitioning when it detects that the optimal partitioning has changed, (2) predicts the scalability of the target program, and (3) calculates the optimal core allocation based on the optimization metric and re-schedules the programs according to it. The detail of (1) through (3) is discussed in Subsections 3.3, 3.2, and in the following paragraphs of this subsection, respectively. Subsection 3.4 describes a key improvement technique to the SBMP scheduler which is called *Core Donation* which aims to maximize the CPU utilization ratio by relaxing the constraint that each core is exclusively used by a single program.

As stated in Section 2, we target in optimizing (minimizing) the average normalized turnaround time or ANTT metric because we believe it is a good metric to represent system performance of multiprogramming workloads. However, the key idea of the proposed scheduler which predicts and utilizes the scalability is not only restricted for optimizing the system against ANTT, but can be used to target different performance metrics such as the weighted speedup metric, if required. This can easily be accomplished by changing the function which seeks for the optimal allocation inside the scheduler.

ANTT is computed as

$$ANTT = \frac{1}{n}\sum_{i=1}^{n} NTT_i = \frac{1}{n}\sum_{i=1}^{n}\frac{C_i^{MP}}{C_i^{SP}} \qquad (1)$$

where C_i^{SP} and C_i^{MP} denote the number of clock cycles to execute a program i under solo and multiprogrammed runs, respectively. Minimizing ANTT means that all the programs are fairly achieving high performance compared to its peak performance (achievable only when occupying

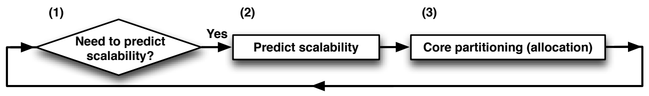

Figure 5: Algorithm overview of SBMP.

the whole system by itself). Therefore, programs that scale almost linearly (such as the applications labeled Green in Figure 4) are removed from the co-scheduling candidate and run in isolation, or gang-scheduling [3].

SBMP scheduler maintains a table called *scalability-table* for each multi-threaded program which consists of pairs of values: each pair consists of a *key* and a *value* where key stores the number of cores to be allocated, and the value stores the relative performance against its peak when executed at the *key* number of cores. Note that having this table is identical with being able to draw the scalability curve shown in Figure 4. The value in the table represents C_i^{MP}/C_i^{SP} in Equation (1), so the scheduler is able to compute ANTT for different core allocations during runtime. The number of possible core assignments is computed as $(N_{core}-1)!/(N_{core}-N_{app})!(N_{app}-1)!$ where $N_{core} > N_{app}$ holds, N_{core} and N_{app} being the number of cores in the system and number of programs that are being scheduled, respectively. It becomes unpractical to calculate and search through all the possible ANTT values to find the optimal allocation when N_{core} and/or N_{app} grow. Therefore, we use a simple hill climbing algorithm similar to the technique used in [8] to find the near optimal assignment.

3.2 Predicting the Scalability

SBMP scheduler needs to know the relative performance against its best solo run for all the co-scheduled programs during runtime to perform its scheduling. We propose to use the number of *cumulative retired instructions* of the program to indicate the performance. That is, we store the cumulative retired instructions per second (IPS) to the *value* of the scalability-table associated with each program. The key notion behind this is the fact that changing the core assignments does not affect the total number of instructions. Figure 6 supports our idea. It shows the number of cores allocated on the x-axis and the number of total instructions on the y-axis for PARSEC benchmarks. All the bars are ob-

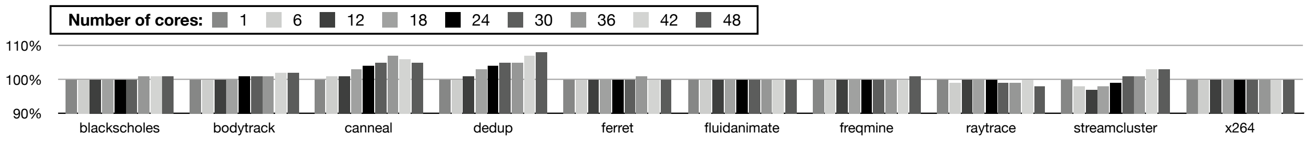

Figure 6: Total number of dynamic instructions of PARSEC benchmarks for different core assignments.

tained by spawning 48 threads, assuming SBMP scheduling. We can see that the applications whose number of instructions varies the most are `canneal` and `dedup` which shows slight variation of up to 8% in the worst case, which is still acceptable for our purpose considering measurement errors. Instruction counts of all the other programs are stable with respect to the number of cores. Therefore, IPS is a perfect indicator to measure the scalability of a program.

As IPS values can be easily obtained by performance monitoring units (PMUs) which come with most of today's microprocessors, it is possible for the scheduler to directly measure the values to fill the scalability-table. For example, the scheduler can dynamically change the allocations during runtime, and run the program for a fixed amount of time to obtain the values. However, we found out that the overhead of measuring the IPS in this manner is not negligible: earning a stable IPS value for a new assignment requires the OS to actually migrate the threads to the new allocation and run for a while to warm up the caches and branch predictors. This requires up to few hundred milliseconds per assignment in the evaluated environment. Recall that we use the processor die in Figure 2 as a minimum unit of allocation, so there are eight (48 divided by six) possible allocations to a single program. Even after careful tuning of the software, it took over three seconds to fill the table, and this overhead will become larger according to the increasing number of cores in the future. Therefore, we introduce a simple model to predict the IPS values. By using the model, we can compute the IPS to fill the scalability-table by sampling the IPS for only a few allocations.

The prediction model we use in this work is the following.

$$\frac{IPS_{N_1}}{IPS_{N_{core}}} = \alpha + \frac{\beta}{N_{core}} + \gamma N_{core} \qquad (\alpha + \beta + \gamma = 1) \qquad (2)$$

IPS_{N_1} and $IPS_{N_{core}}$ indicate the IPS when allocated a single core and N cores, respectively. The left side of the equation represents the relative turnaround time against single core execution, and the left two terms of the right side is the same as that of Amdahl's law equation [1]. Recall that Amdahl's law is a simple yet powerful model that models the scalability of a parallel program. The formula is quite intuitive and the scalability curve is known to fit well with real world programs. However, as can be seen from Figure 4, the scalability of some applications tend to not only saturate at some point but also degrade its performance by adding more number of cores. This cannot be represented by the original Amdahl's law. Therefore, we added the last term γN_{core} which we expect to represent some overhead associated with adding more number of cores. The condition equation in the parentheses follows the original condition of the Amdahl's law that the sum of the coefficients equals one, and this is because we assume that the parallel portion of the program represented as β in the original equation, can be divided into two sections where one can be parallelized ($\frac{\beta}{N_{core}}$) and the other adds an overhead (γN_{core}). Therefore, we set the condition as the sum of α, β and γ equals one.

To obtain the values of the coefficients, we use the least squares approach. Therefore, we need to at least collect the IPS with three configurations (two different configurations in addition to single core execution) in order to obtain the three coefficients because Equation (2) has two individual parameters.

3.3 Re-partitioning Algorithm

As seen from Figure 5, SBMP scheduler tries to change the core partitioning when it considers that the optimal allocation has changed during the multiprogramming execution. There are mainly two events that invokes the SBMP scheduler to re-partition during execution. Those events are (1) creation or termination of a program and (2) phase transition detected in any of the programs.

The first is natural, because it means that the multiprogramming workload has changed, and thus there is a chance to find a better partitioning. When any of the program finishes its execution, SBMP simply calculates the optimal allocation using the existing scalability-table and allocates the cores according to it. When a new program is created, then SBMP scheduler predicts the scalability of it as described in Subsection 3.2, and performs the partitioning.

The second is more aggressive which utilizes the information obtained online to dynamically find a better core assignment. Programs are typically composed of different execution phases which show different characteristics, and SBMP scheduler attempts to keep up with the optimal core partitioning. Researchers have tried to detect phases to perform dynamic optimization, and there exists different definition of phases according to the optimization techniques. In this paper, we consider phase as a region of program having different scalability.

Figure 7 shows an example execution trace of `x264` showing the execution phases having different scalability. x-axis and y-axis of Figure 7 (a) represents time in milliseconds and the cumulative execution cycles, respectively. The thick dotted line on top of the figure shows the maximum available cumulative cycles which is equivalent to the cumulative execution cycles when the CPU is 100% utilized. The cumulative cycles is sampled every 50 ms. CPU utilization ratio is obtained by dividing the cumulative execution cycles by the maximum available cumulative cycles. We can see from the figure that the trace can be divided into three regions or phases by focusing on the cumulative execution cycles (identical to CPU utilization), which we named phase A, B and C. Three figures of Figure 7 (b) on the right side show the scalability for those three phases. X-axis indicates the number of cores allocated and the y-axis shows the relative performance against a single-core execution. We can see similar trends for phases A and C which show a gentle curve giving the maximum performance at the maximum number of cores, and a much steeper scalability curve for phase B showing the

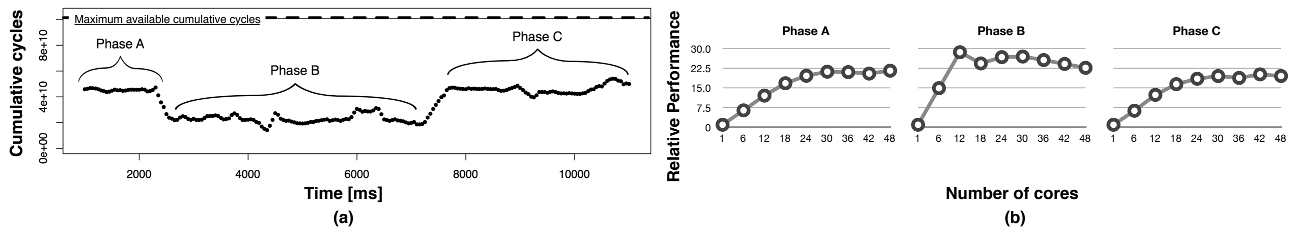

Figure 7: Execution trace of x264 showing the relationships between CPU utilization ratio and scalability. (a) Time series plots of cumulative cycles and (b) scalability of different points are shown.

Table 1: CPU utilization ratio (%) of PARSEC benchmarks

# of cores	bl	fl	fr	fe	x2	ra	bo	de	ca	st
6	98.6	96.7	96.4	99.9	99.5	89.0	73.7	73.9	82.1	92.6
12	93.6	90.4	86.8	92.7	97.2	78.4	55.5	59.0	71.1	80.7
18	89.8	83.4	65.4	95.4	83.9	71.9	46.4	51.1	68.5	67.3
24	85.3	74.8	68.6	87.9	72.2	62.9	39.0	44.8	69.8	56.7
30	84.4	68.0	61.3	74.7	59.3	60.2	32.5	38.8	67.3	50.6
36	80.5	62.0	43.1	82.2	46.1	50.5	29.9	34.9	66.2	45.7
42	69.6	58.8	34.9	72.6	38.7	43.8	26.8	31.4	70.4	42.3
48	85.7	54.5	44.5	72.1	33.3	37.6	24.1	28.7	65.9	37.7

highest performance at 12 cores. These graphs motivate us to detect the phase changes online and dynamically perform a re-partitioning. Also, we can see that the CPU utilization ratio which can be obtained from cumulative execution cycles is a good indicator to detect phase transition. Therefore, SBMP scheduler detects the phase change by monitoring its cumulative execution cycles. SBMP scheduler monitors the PMUs every *epoch* (or time window), and whenever a change of the counter values greater than a certain *threshold* compared to the previous epoch has been observed, SBMP scheduler enters the scalability prediction process to perform re-scheduling.

3.4 Core Donation

The design of SBMP scheduler is described in the above subsections, and here we introduce *Core Donation*, a sophisticated technique to improve the system performance in SBMP scheduler. As we have shown in Figure 7, different programs have different CPU utilization ratios. Table 1 shows the average CPU utilization ratio of PARSEC benchmarks (benchmark names are represented with their first two letters) for different number of cores being allocated. We can see a trend across benchmarks that by allocating more number of cores, the CPU utilization tends to decrease. This is a fundamental problem in multi-threaded applications which does not have linear scalability, so we have to overcome this tendency to achieve high performance. In addition, there are variability in the utilization ratio that some programs have high utilization (`blackscholes`, `ferret`, and `fluidanimate`) while others have poor utilization (`bodytrack` and `canneal`). One of the most important aspects of the OS scheduler is to keep the CPU as busy as possible by continuously assigning threads that are ready to execute. However, SBMP scheduler partitions the processor, and only a single program is allowed to run on each core which tends to waste the precious CPU time.

We provide a solution to this problem by letting a program which has a utilization ratio lower than a certain threshold to be a "donor" of CPU cores (or CPU time). After allocating the cores to each program, SBMP scheduler checks if any of the program meets the requirement of a donor. If it finds a correspondent program, SBMP scheduler picks another program which benefits the most by getting additional cores (which has a high utilization ratio and good scalability) from the scalability-table, and sets the program as a "donee". SBMP scheduler allows the donee program to be executed on the donor's cores by allocating the program to its original partition and also the donor's partition. Therefore, there are two multi-threaded applications which can run on donor's cores. Donee is kept from interfering with the donor's threads by executing its threads only when the donor cannot utilize the CPU. We use the thread priority mechanism to achieve this requirement by making the priority of donee program much lower than that of the donor. This makes the situation where donor can preempt donee's execution while the opposite is not possible. Core Donation is applied after core partitioning (see (3) of Figure 5).

The interesting point of Core Donation is that it works in synergy with the SBMP scheduler by optimizing against different granularities. SBMP scheduler assigns the cores to programs by using the processor die as a minimal unit, which is six cores in the evaluation platform. This is rather a coarse grain spatial optimization while Core Donation tries to fill in the gap and apply finer granularity optimization after a rough optimization done by SBMP scheduler. Also, Core Donation works in a much finer time granularity than the original SBMP, and is possible to increase the CPU utilization to improve system performance. Therefore, by applying the partitioning of SBMP scheduler and the optimization of Core Donation, the proposed set of techniques make the most out of manycore architectures to achieve high performance.

We have to be careful that the utilization ratio to detect a phase change for donee programs should be only measured at its original partition (there might be some disturbance by the donor program at the donor's cores). Note that each program is allowed to either be a donor or donee but not both, and a donor is only allowed to have one donee.

4. EXPERIMENTAL SETUP

4.1 Hardware Platform

We perform experiments on a quad socket IBM System x3755 M3 server which consists of four 12-core AMD Opteron 6172 microprocessors running at 2.1 GHz forming a 48-core system. Each socket integrates two six-core dies with each core having its own private L1 and L2 caches along with a shared 12 MB L3 cache. Table 2 shows relevant hardware parameters of the evaluation platform.

Table 2: Parameters of the evaluation platform

Processor:	4 × AMD Opteron 6172
# of dies per processor	2
# of cores per die:	6
Total # of cores:	48
L3 cache size:	12 MB per socket
Main memory:	96 GB DDR3 PC3-10600

Table 3: Parameters of the SBMP scheduler

Threshold of phase change detection:	Utilization ratio more than doubles or gets smaller than half of the previous epoch
Length of epoch:	2.5 seconds
Length of the first epoch after re-scheduling:	10 seconds
# of samples for prediction: (# of cores)	four points (1, 12, 24 and 48)
Overhead of prediction:	≈ 550 milliseconds
Threshold of becoming a "donor" in Core Donation:	70%

4.2 SBMP Scheduler Implementation

We have implemented a prototype of SBMP scheduler as a user level software. The evaluation system runs Linux kernel 2.6.37.6, and the modified version of `perf-tools` toolset is used to allow periodical access to the PMUs. CPU clock frequency scaling is disabled to avoid measurement variance. We use standard Linux API (`sched_setaffinity(2)` and `setpriority(2)`) to control the CPU affinity of processes to bind the programs to specific cores and to manage the priority (niceness) of each program in order to apply Core Donation technique.

Detailed values of parameters used inside the SBMP scheduler are shown in Table 3. Threshold to detect phase change and its length of sampling epoch is experimentally obtained by taking the overhead of prediction into account. Also, we set the first epoch after re-scheduling to 10 seconds to prevent oscillation of re-scheduling. For the sampling points, increasing the number of samples tends to increase the fitting accuracy but the associated overhead of sampling also becomes larger (≈150ms for each sample). Therefore, we constructed a model of all the possible combinations for each benchmark in advance and compared the fitting accuracy to the plot shown in Figure 4. As a result, four sample points with 1, 12, 24 and 48 cores gave us a good balance between fitting accuracy (coefficient of determination over 0.99) and the overhead on average and we chose them as shown in the table. We apply Core Donation if the utilization of any of the programs being executed is below 70%.

4.3 Workloads

We use PARSEC benchmark suite 2.1 [4] to evaluate SBMP scheduler against the Linux default scheduler. PARSEC benchmark suite is designed to evaluate shared-memory computers, and is composed of a set of emerging workloads for future manycore processors. We compile PARSEC with GCC-4.1 with "`-O3 -funroll-loops -fprefetch-loop-arrays`" options. We use *native* input sets for all benchmarks except `dedup`. Because the execution time of `dedup` was too short compared to other programs, we use a larger input (the Fedora 16 x86_64 DVD ISO).

We have shown 10 out of 13 PARSEC benchmarks in figures 4 and 6 where `facesim`, `swaptions` and `vips` are excluded because we could not correctly compile them on the evaluation platform. Additionally, we exclude `blackscholes` and `fluidanimate` which are categorized as Green in Figure 4. Highly scalable applications prefer being executed in a gang-scheduling manner as shown in other work [3], where SBMP scheduler can detect these applications as having good scalability by checking the scalability-table, and remove from co-scheduling in order to perform gang-scheduling. Although categorized as Green, we left `freqmine` in the evaluation workloads because it has a reduction operation which shows multiple phase changes during its execution and is used to show the effectiveness of our phase detection capability.

We evaluate SBMP scheduler with variety of workloads where each workload is composed of four benchmark programs. SBMP scheduler is designed to optimize the scheduling under multiprogramming environment with a number of parallel programs with wide varieties of scalability, however, increasing the number of applications within each workload reduces the possible allocations resulting in no space to optimize. For example, as we give each program at least six cores, having more than four programs within a workload results in giving at least one program to only stick on the minimum allocation of six cores. Therefore, we chose four as a number to create each workload because it gives us a good balance between the optimization flexibility as well as enough load on the target system.

Because the main target of SBMP scheduler is efficient co-scheduling of parallel regions of multi-threaded multiprogramming workloads, we use Berkeley Lab Checkpoint/Restart (BLCR) tools [13] to set up an environment which enables *checkpoint and restart* of programs to evaluate only the parallel region called the *region of interest (ROI)* of PARSEC, which is marked in the original source code. Our evaluation is conducted using this system running from the beginning till the end of ROI for all benchmarks.

The evaluation methodology is similar to the one originally proposed for SMT job scheduling [26], and a number of following work have been evaluated in a similar manner [2, 12, 15, 29]. To account for the varieties of execution times (from tens to hundreds of seconds), we restart an application instantaneously when it finishes execution until all the programs are executed at least three times to completion. We measure the execution time of each application using the `time` command in Linux.

5. EVALUATION RESULTS

5.1 Programs with a Single Phase

First, we show the results with the workloads which does not include applications that have multiple phases within programs. For this purpose, we have set up six workloads which include two applications classified as Red (`canneal` and `streamcluster`), and the other two from Yellow benchmarks (`bodytrack`, `dedup`, `ferret` and `raytrace`). The Red applications have low scalability and thus assigning a large number of cores will hurt system performance. Therefore, we can see if SBMP scheduler can accurately predict the scalability of the program. Evaluated workloads are shown in Table 4.

Figure 8 shows the ANTT for each workload of *SBMP-base*, which is the simplest implementation of SBMP scheduler which does not have the ability of phase prediction nor

Table 4: Multiprogrammed workloads to evaluate SBMP-base

ID	Benchmarks	Type
1	bo-ca-de-st	YYRR
2	bo-ca-fe-st	YYRR
3	bo-ca-ra-st	YYRR
4	ca-de-fe-st	YYRR
5	ca-de-ra-st	YYRR
6	ca-fe-ra-st	YYRR

Table 5: Multiprogrammed workloads to evaluate SBMP-PP

ID	Benchmarks	Type	ID	Benchmarks	Type
7	bo-ca-fe-fr	GYYR	16	de-fr-st-x2	GYYR
8	bo-ca-fr-x2	GYYR	17	fe-fr-ra-st	GYYR
9	bo-fe-fr-st	GYYR	18	fr-ra-st-x2	GYYR
10	bo-fr-st-x2	GYYR	19	bo-ca-fe-x2	YYYR
11	ca-de-fe-fr	GYYR	20	bo-fe-st-x2	YYYR
12	ca-de-fr-x2	GYYR	21	ca-de-fe-x2	YYYR
13	ca-fe-fr-ra	GYYR	22	ca-fe-ra-x2	YYYR
14	ca-fr-ra-x2	GYYR	23	de-fe-st-x2	YYYR
15	de-fe-fr-st	GYYR	24	fe-ra-st-x2	YYYR

Table 6: Multiprogrammed workloads to evaluate SBMP-CD

ID	Benchmarks	Type	ID	Benchmarks	Type
25	bo-de-fe-fr	GYYY	32	bo-de-ra-x2	YYYY
26	bo-de-fe-x2	GYYY	33	bo-ca-de-fe	YYYR
27	bo-de-fr-ra	GYYY	34	bo-ca-de-ra	YYYR
28	bo-de-fr-x2	GYYY	35	bo-ca-de-x2	YYYR
29	bo-ca-de-fr	GYYR	36	bo-de-fe-st	YYYR
30	bo-de-fr-st	GYYR	37	bo-de-ra-st	YYYR
31	bo-de-fe-ra	YYYY	38	bo-de-st-x2	YYYR

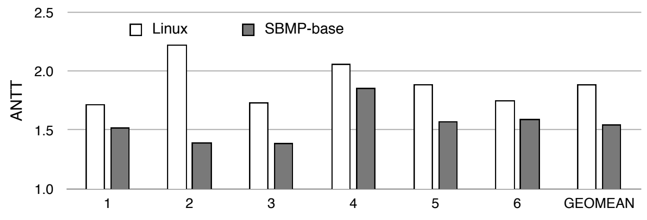

Figure 8: ANTT of Linux and SBMP-base schedulers for applications with a single phase.

Core Donation, is compared against Linux scheduler. Average ANTT of Linux scheduler is 1.88, and SBMP-base whose average ANTT is 1.54 well outperforms Linux scheduler for all the workloads evaluated. The results show that effectively choosing the core allocation based on the scalability of each program gives performance benefit over Linux scheduler. From these results, we can conclude that the proposed scalability prediction algorithm described in Subsection 3.2 succeeds in predicting the scalability of programs.

5.2 Programs with Multiple Phases

Next, we present the results by evaluating the workloads which contain applications that have multiple phases within the program such as `x264` shown in Figure 7. The evaluated workloads are shown in Table 5. All the workloads include two out of three applications which have multiple phases (`freqmine`, `ferret` and `x264`), one from Yellow applications, and another from Red applications.

Figure 9 shows the result of the Linux scheduler, SBMP-base and *SBMP-PP* which is SBMP-base with phase prediction enabled. Average ANTT of the Linux scheduler, SBMP-base and SBMP-PP are 1.89, 2.09 and 1.77, respectively. We can see that while SBMP-base shows lower performance than the Linux scheduler, SBMP-PP significantly outperforms both, showing the superiority of SBMP-PP. The representative contribution of SBMP-PP can be seen in many workloads such as 10, 11, 12, 13 and so on, where SBMP-base has the worst case ANTT. Because SBMP-base only predicts the scalability of each program at the point of creation/termination, it tremendously suffers from making a wrong decision at the point of re-scheduling. However, because SBMP-PP has the ability to dynamically detect phase transition, it can recover from making a bad decision. This reveals us that the aggressive re-scheduling based on phase detection is beneficial for programs which are composed of multiple phases. We have presented that detecting and optimizing against the scalability, which is the key insight of SBMP scheduling, is effective in scheduling multi-threaded multiprogramming workloads.

5.3 Programs with Various CPU Utilizations

Now, we show whether Core Donation can improve performance against the Linux scheduler and SBMP-PP. The 14 workloads which are evaluated are presented in Table 6. These workloads are comprised of two applications which have low CPU utilization (`bodytrack` and `dedup` as can be seen from Table 1), and all the other applications.

Figure 10 shows the performance results of the Linux scheduler, SBMP-PP and *SBMP-CD* which utilizes Core Donation technique along with SBMP-PP. The average ANTT among the workloads of the Linux scheduler, SBMP-PP and SBMP-CD are 2.06, 1.68 and 1.60, respectively. SBMP-CD shows the best performance on average across the workloads. Because the workloads are composed of applications which have low CPU utilization, the Linux scheduler has the advantage of effectively utilizing the CPUs by assigning ready-to-execute threads one after another for some workloads. However, SBMP-CD performs better than the Linux scheduler by utilizing the CPU time given by the donor applications. This is well confirmed by comparing SBMP-PP against SBMP-CD, which shows that SBMP-CD achieves better or at least comparable performance over SBMP-PP. This shows that the idea of donors giving the CPU time to donees without interfering donor's threads gives performance improvement. Although ANTT difference between SBMP-PP and SBMP-CD is not so large compared to other results (Linux and SBMP-base in Figure 8, and SBMP-base and SBMP-pp in Figure 9), we believe that by having more number of cores on the system and broader optimization space will widen this gap between SBMP-PP and SBMP-CD. Also, another optimization to be considered is that a donor having more than one donees at the same time, which is left for future work.

5.4 Overall Performance with All Programs

Figure 11 presents the ANTT comparison of all the schedulers for all 70 workloads we have evaluated. The figure shows the workloads on the x-axis and the ANTT on the y-axis. We present the curves independently sorted for each scheduler. The overall average ANTT for the Linux, SBMP-base, SBMP-PP and SBMP-CD schedulers are 1.83, 1.99,

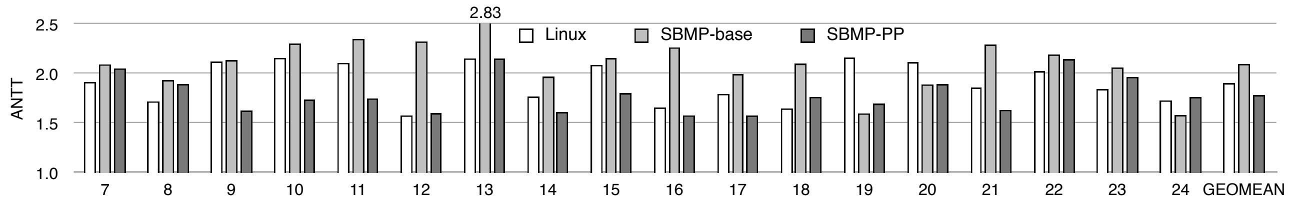

Figure 9: ANTT of Linux and SBMP-PP schedulers for applications with multiple phases.

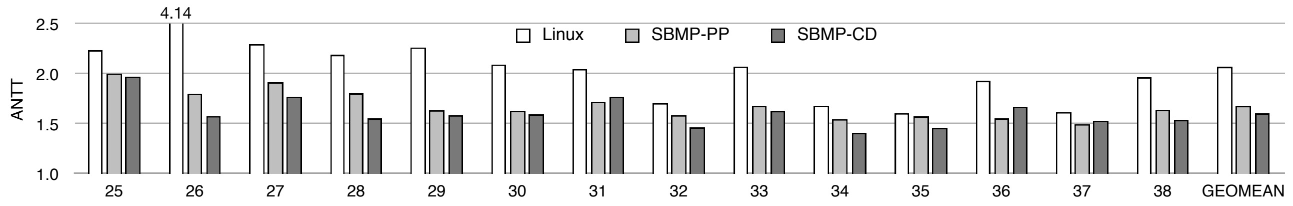

Figure 10: ANTT of Linux and SBMP-CD schedulers for applications with low CPU utilization.

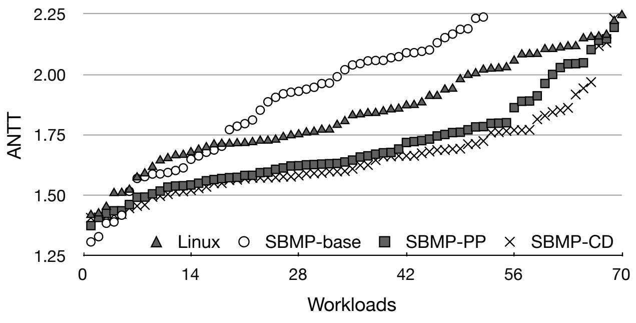

Figure 11: ANTT of all workloads.

1.70 and 1.65, respectively. As can be seen from the figure, SBMP-base scheduler shows the worst performance, and shows 8% degradation against the Linux scheduler. Interestingly, the lowest ANTT values are achieved by SBMP-base, showing the superiority of it when there is only a single phase within the applications. By adding the capability of phase prediction and adaptation to SBMP-base, performance significantly improves and outperforms the Linux scheduler for 8%. Finally, SBMP-CD shows better ANTT over SBMP-PP for 3%, which can clearly seen from the right hand side of figure.

6. RELATED WORK

Scheduling for multiprogrammed multi-threaded applications: The work most close to ours is the HOLISYN schedulers by Bhadauria and McKee [3]. The base concept of applying dynamic space-sharing for applications that do not scale is the same, however, they aim at dealing with scalability limitation caused by hardware resource contention, and the optimization target of their work is energy delay product, and therefore the policy of deciding the number of cores to allocate differs from our work. Our technique considers the scalability of a program limited by the software nature by predicting its scalability, and evaluated with a 48-core system to show the effectiveness. Dynamic space-sharing techniques are well studied in the HPC area [10, 25], and approaches to dynamically adjust the scalability of a program is also examined [9]. Recent work called Thread Tailor [16] proposes a dynamic compilation system that can automatically adjusts the number of threads by combining multiple threads together.

Avoiding contention through scheduling: OS scheduling approaches to avoid contention in multicore processors have been studied as well [12, 22, 30]. Contention in shared resources other than LLC such as the memory controller, prefetcher or the memory bus is also a big cause of performance degradation. In some study, LLC miss rate is shown to be a good indicator of shared resource intensiveness and sensitiveness [30]. A cache model which predicts the performance impact of process-to-core assignment is used in a user level scheduler to improve performance [2]. Our work differs from these studies in several ways but the most important point is that we target multi-threaded applications while they concentrate on single-threaded applications.

NUMA aware systems: Recently, several work have focused on optimizing process scheduling on NUMA systems by taking the physical locations of cores, LLC, and memory controller into account. A simple model proposed to characterize the local and remote memory system performance of NUMA system has been evaluated [20]. In NUMA systems, even when LLC competing threads are on the same die, the optimal scheduling might not be separating these threads apart because a thread being migrated will incur higher latency for memory accesses [6, 19]. The shared resource contention as well as data locality should be considered together to find the optimal scheduling. Our platform is also a NUMA and NUCA system, however, SBMP scheduler has not focused on thread migration to reduce shared resource contention. SBMP scheduler applies a centralized thread assignment which minimizes the shared resource contention at the memory hierarchy in nature. However, we have not evaluated throughly by considering NUMA characteristics and we believe there is still room left for optimization. Those are left for future work.

7. CONCLUSIONS AND FUTURE WORK

In this work, we investigate the OS scheduling for the manycore era where various parallel applications with diverse scalability are co-scheduled. Traditional OS scheduler maintains fairness by giving fair amount of CPU time to each thread or process, however we believe that there are two key factors that the scheduler has to be concerned: scalability and CPU utilization. We have shown that scalability of applications should be taken into account to achieve high system performance, where the key idea is to allocate the appropriate amount of shared resources to programs according to their scalability, in order to maintain high system performance. We have proposed a technique to dynamically predict the scalability of the program, as well as a sophisticated phase recognition technique. We have also found that CPU utilization of multi-threaded applications have diverse characteristics, and proposed *Core Donation* technique which tries to maximize the CPU utilization of the system while still keeping the scalability into account.

We have built a prototype SBMP scheduler and compared the performance with the Linux scheduler for variety of workloads. SBMP scheduler shows higher system performance compared to the Linux scheduler, indicating each optimization technique presented in the paper is quite effective. We have not done any optimizations to avoid shared resource contention, and there might be better scheduling when carefully considering the NUCA and NUMA architecture characteristics in detail. Exploration of the scheduler on NUCA and NUMA architectures are left for future work.

8. ACKNOWLEDGMENTS

This research was supported in part by New Energy and Industrial Technology Development Organization, and JSPS Grant-in-Aid for Challenging Exploratory Research (22650008).

9. REFERENCES

[1] G. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring)*, 1967.

[2] M. Banikazemi et al. PAM: a novel performance/power aware meta-scheduler for multi-core systems. In *SC '08*, 2008.

[3] M. Bhadauria and S. McKee. An approach to resource-aware co-scheduling for CMPs. In *ICS '10*, 2010.

[4] C. Bienia. *Benchmarking modern multiprocessors.* PhD thesis, Princeton University, 2011.

[5] S. Blagodurov et al. Contention-aware scheduling on multicore systems. *ACM Transactions on Computer Systems (TOCS)*, 28(4), 2010.

[6] S. Blagodurov et al. A case for NUMA-aware contention management on multicore systems. In *USENIXATC '11*, 2011.

[7] R. Cochran et al. Pack & Cap: adaptive DVFS and thread packing under power caps. In *MICRO 44*, pages 175–185. ACM Request Permissions, Dec. 2011.

[8] J. Corbalán et al. Performance-driven processor allocation. In *OSDI '00*, 2000.

[9] J. Corbalán et al. Improving Gang Scheduling through job performance analysis and malleability. In *ICS '01*, 2001.

[10] J. Corbalán et al. Performance-driven processor allocation. *IEEE Transactions on Parallel and Distributed Systems*, 16(7):599–611, 2005.

[11] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, 2008.

[12] A. Fedorova et al. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *PACT '07*, 2007.

[13] P. Hargrove and J. Duell. Berkeley lab checkpoint/restart (BLCR) for linux clusters. *Journal of Physics: Conference Series*, 46:494, 2006.

[14] J. Huh et al. Exploring the Design Space of Future CMPs. In *PACT '01*, 2001.

[15] R. Knauerhase et al. Using OS observations to improve performance in multicore systems. *IEEE Micro*, 28(3), 2008.

[16] J. Lee et al. Thread tailor: dynamically weaving threads together for efficient, adaptive parallel applications. In *ISCA '10*, 2010.

[17] J. Lin et al. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *HPCA '08*, pages 367–378, 2008.

[18] K. Lun et al. Balancing thoughput and fairness in SMT processors. In *ISPASS '01*, 2001.

[19] Z. Majo and T. Gross. Memory management in NUMA multicore systems: trapped between cache contention and interconnect overhead. In *ISMM '11*, 2011.

[20] Z. Majo and T. Gross. Memory system performance in a NUMA multicore multiprocessor. In *SYSTOR '11*, 2011.

[21] C. Pheatt. Intel® threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4), 2008.

[22] K. Pusukuri et al. FACT: a framework for adaptive contention-aware thread migrations. In *CF '11*, 2011.

[23] M. Qureshi and Y. Patt. Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO 39*, 2006.

[24] K. Sankaralingam et al. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *ISCA '03*, 2003.

[25] C. Severance and R. Enbody. Comparing Gang Scheduling with dynamic space sharing on symmetric multiprocessors using automatic self-allocating threads (ASAT). In *IPPS '97*, 1997.

[26] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *ASPLOS-IX*, 2000.

[27] M. Taylor et al. The raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2), 2002.

[28] J. Weinberg and A. Snavely. User-guided symbiotic space-sharing of real workloads. In *ICS '06*, 2006.

[29] S. Zhuravlev et al. Addressing shared resource contention in multicore processors via scheduling. In *ASPLOS '10*, 2010.

[30] S. Zhuravlev et al. AKULA: a toolset for experimenting and developing thread placement algorithms on multicore systems. In *PACT '10*, 2010.