PAM: A Novel Performance/Power Aware Meta-scheduler for Multi-core Systems

Mohammad Banikazemi Dan Poff Bulent Abali
IBM Thomas J. Watson Research Center
Hawthorne, NY
Email: {mb,poff,abali}@us.ibm.com

***Abstract*—Sharing resources such as caches and main memory bandwidth in multi-core systems requires a more sophisticated scheduling scheme. PAM is a low-overhead, user-level meta-scheduler which does not require any hardware or software changes. In particular, it operates by detecting resource congestions and providing guidelines to the standard system scheduler by limiting the assignment of processes to subsets of available cores. PAM contains a cache model that it uses to predict the impact of new schedules. PAM can be used to improve the system along three dimensions: performance, power, and energy consumption (and any combination of these three). On our prototype, we show individual benchmarks can improve by up to 33% and the overall system performance can be improved by as much as 14%.**

I. Introduction

As multi-core and multi-chip servers are becoming widely used, it is becoming necessary to exploit architectural characteristics of these systems in order to get the highest performance. One of the major characteristics of such systems is sharing of resources such as caches, buses, and memory bandwidth among cores and chips. Increased power consumption is also becoming a problem with ever increasing chip density and speed. State of the art operating systems (OS) typically do not exploit architectural characteristics of these systems (or use very basic and limited methods such as those used by Linux). Many OSes are neither power nor energy aware, and they receive little or no feedback from hardware on overall system performance. In this paper we address the problem of performance and power aware scheduling in multi-core systems. Our work can also be easily extended to produce energy aware scheduling as well.

We present the design and implementation of a Performance/power Aware Meta-scheduler called PAM. Using PAM we are able to dynamically optimize multi-core servers along performance and power (and energy) dimensions. Results for industry standard benchmarks and server platforms presented in this paper show double digit improvements in these dimensions.

PAM monitors performance (e.g., instructions per cycle and cache miss ratio), power, and energy by accessing the hardware performance counters and power monitoring (PM) hardware. It should be noted that if PM hardware does not exist on the platform, there are means to accurately estimate power from hardware performance counters. Based on this information, PAM gives directions to the OS scheduler (using existing OS interfaces) to re-map software threads to the hardware threads in anticipation of lower power or energy consumption or higher performance. PAM runs in the user space. It requires no changes to the OS kernel or the workloads. Re-mapping software threads in PAM may take many forms, such as separating high cache consuming software threads from each other. In the power and energy optimization scenarios, software threads may be scheduled on fewer hardware threads to reduce power/energy consumption. These methods are orthogonal to the traditional power management techniques, such as voltage and frequency scaling and therefore can be employed in addition to them.

PAM meta-scheduler uses a simple two-state state machine with a cycle time of once a second and it can dynamically track and schedule any workload. To remap software threads to hardware threads we take advantage of the *cpusets* built in to the Linux kernel, although other interfaces for binding processes to CPUs may be used as well.

Main contributions of this paper are:

- We describe the design and implementation of a practical user-space meta scheduler called PAM for optimizing system power, performance and energy in multi-core systems which (a) requires no changes to the OS, (b) works on existing industry standard platforms and uses existing hardware performance counters, (c) works with any unmodified application binary, and (d) does not require architecture simulation, tracing, profiling or re-running of the workloads.
- We describe an algebraic model of multi-core systems, modeling a hierarchy of processors, shared

SC2008 November 2008, Austin, Texas, USA 978-1-4244-2835-9/08 $25.00 ©2008 IEEE

caches, and memory. The model is very simple yet powerful for estimating a software thread's performance, footprint and miss rate individually with shared caches. We describe an algorithm based on this model for producing good performing schedules (i.e., mapping of n software-threads to n hardware-threads).
- We ran SPEC CPU2006 benchmarks on an IBM HS21XM blade server (Intel Xeon E5345 processor, 2.33GHz four cores per socket, two sockets, with four 4MB L2 caches) along with PAM giving scheduling instructions to the Linux OS. Relative to the base Linux scheduler, we observed execution time reduction of as much as 33% for individual threads and up to 14% system-wide, demonstrating the benefits of our scheduler design and implementation.
- On our experimental system we demonstrate dynamic power management and power capping capability via scheduling only, namely by regulating the number of processors as a function of power consumption of the workloads.

The rest of the paper is organized as follows. Background is discussed in Section II. The basic ideas behind PAM are presented in Section III. A more comprehensive discussion of PAM along with new algorithms for estimating the impact of new schedules are presented in Section IV. Section V contains the performance evaluation of PAM. Related work is discussed in Section VI. We end the paper with conclusions and future work in Section VII.

II. Preliminaries

In this section we first discuss hardware counters and how they can be used to monitor the state of a system. We then discuss the Linux *cpusets* which we use in our implementation.

A. Monitoring performance

Through hardware performance counters existing on many modern processors, it is possible to monitor the execution of unmodified executables. Periodically sampled performance metrics such as the number of retired instructions and L1 and L2 cache miss ratios reveal important details about the applications running on a processor. PAM uses the hardware performance counters, first to make rational scheduling decisions, and second to measure the effectiveness of those decisions by comparing the before- and after-scheduling values of counters. We used the Pfmon2 open source package to collect the hardware counter information [1], [2]. Pfmon2 is available for Linux however one can gather the values of hardware counters on any OS with appropriate device drivers.

For the purposes of scheduling we monitor several events including: cycles per instruction (CPI), ratio of L1 data cache misses to L1 data references (miss ratio m_1), ratio of L2 unified cache misses to references (miss ratio m_2), L2 prefetch counts, and floating point instructions issued. CPI metric is a measure of the execution efficiency of an application; lower the CPI faster the execution. CPI is a time variant and application dependent metric regularly sampled in our scheduling scheme. When the processor execution stalls, for example due to L1/L2 misses, the CPI value will increase since it's taking longer for the processor to execute. The Instructions Per Cycle (IPC) metric is calculated as $\frac{1}{CPI}$ and is a measure the throughput of a CPU. On a multiprocessor system, we need a metric for the system throughput. Using the $\sum IPC$ or $\sum CPI$ metrics with dissimilar workloads (with different inherent CPI values) may not treat the workloads fairly and it has other problems as explained in [3]. Time to completion is a better way to quantify multiprocessor performance. For our purposes here, we define the *Speedup* metric to measure effectiveness of our scheduling algorithms and actions:

$$Speedup = \frac{CPI_{\text{before}}}{CPI_{\text{after}}},$$

$$System\ Speedup = \left[\prod_{i}^{n} Speedup_i\right]^{\frac{1}{n}} \quad (1)$$

where n is the number of processors. Speedup simply indicates the amount of increase or decrease in execution speed ($Speedup = 1$ means no change). We measure the CPI values before and after a scheduling action, and if $Speedup > 1$ it means that the action has been a profitable one. For quantifying the total system speedup on n processors we use the geometric mean of n speedup values. When reporting benchmark results in Section V we again used the speedup formula however this time using benchmarks total execution time instead of CPI values.

B. Monitoring power

Recent IBM rack mount and BladeCenter servers have built-in power monitoring hardware (PM). PM hardware samples the power and adds them in to the Energy Accumulation Register (EAR) at millisecond intervals. Power measured is for the whole server, including not only the CPU but all the memory and circuits in the server [4], [5]. We used the IPMI interface on the server to bring out the EAR register for PAM meta-scheduler to

monitor power. In the absence of built-in power monitoring hardware, other approaches to power measurement exist such as using *Intelligent Power Distribution Units* or correlating hardware performance counters to power utilization that we won't detail here.

C. Cpusets

For binding processes to CPUs we use the notion of a *cpuset*, defined as a set of CPUs that a process may be bound to. On an N processor system there are $2^N - 1$ possible cpusets including the maximal set of the N CPUs (the *default* cpuset). For example, on the 8 processor system shown in Fig. 1a, we may define a set of cpusets called $L2separate$ consisting of two cpusets { {0,2,4,6}, {1,3,5,7} } each of which emphasizes L2 exclusion as shown in Fig. 1b. We may also define a set of cpusets called $L2sharing$ consisting of { {0,1}, {2,3}, {4,5}, {6,7} } each of which emphasizes the sharing between the CPUs. For example, if we want two processes A and B not to share an L2 cache because one or both may have a large L2 footprint, we then put them on an $L2separate$ cpuset, for example bind processes A and B to the $cpuset = \{0,2,4,6\}$, which guarantees that they will end up using separate L2 caches to avoid L2 contention. Likewise, if A and B have some affinity to each other, for example by having shared variables, we can bind them to on an $L2sharing$ cpuset. In another example, we can define power/energy aware cpusets, such as { {0,1,2,3,4,5,6,7}, {0,1,2,3,4,5}, {0,1,2,3}, {0,1} } each of which will consume varying degrees of power and energy depending on the number of CPUs in a cpuset.

By using these cpusets with well-defined characteristics (a) we exploit the geometry of the hardware organization, and (b) we make only high level process-to-cpuset mapping decisions and we let the OS scheduler decide for the exact mapping of processes to CPUs (within a cpuset). Note also that the cpuset mechanism allows us to recursively define subsets of a cpuset. The sub-cpusets allow us to do multi-level performance, power, energy optimizations. For example, once an L2 based cpuset has been chosen for a set of processes, we can then make L1 cache optimizing decisions such as running those processes on the same or separate L1 caches, or making power/energy decisions within the chosen L2 based cpuset.

For convenience of implementation, we used the /dev/cpuset pseudo filesystem already found in the Linux kernel [6]. However, any method equivalent to the $sched\_setaffinity()$ system call for binding a process to a set of CPUs may be used on any operating system.

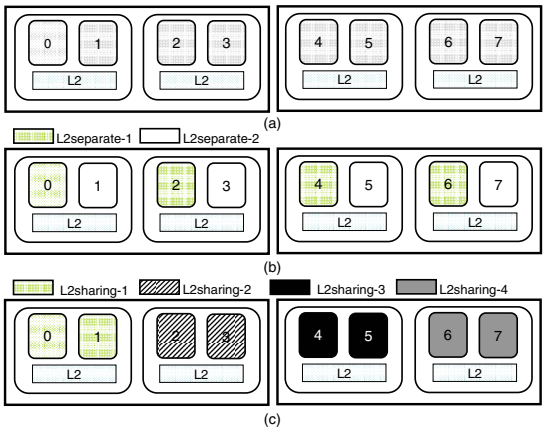

Fig. 1. A system with eight cores: two sockets, two chips per socket, and two cores per chip, and several cpusets highlighted.

III. Basic Idea

PAM relies on detecting resource bottlenecks and/or undesired conditions by monitoring hardware performance counters. After detecting such a condition, PAM guides the system scheduler to reschedule processes (or software threads) such that contention for resources is minimized. An important characteristic of PAM is that it then monitors the system to decide whether the new schedule has been beneficial or not (by using metrics later defined in this section). If improvement is detected, system stays with the new schedule for some amount of time. If not, the default schedule is used. PAM performs these operations continuously. Even though PAM can be implemented as part of the OS kernel, it can also be implemented as a tool without requiring any changes in the system software or direct interaction with system scheduler. We first explain the PAM execution model and then present various conditions it detects.

A. PAM Execution Model

PAM uses a simple 2-state logic in making scheduling decisions (Fig. 2). The benefit of this scheme is that (a) it is simple to implement requiring no changes in the OS source code (i.e. the Linux kernel scheduler) although it does not exclude a kernel based implementation, (b) by design it produces schedules which perform no worse than that of the base OS scheduler by a user defined margin.

The two states are named *Default* and *Good*. Default is an observation state and Good is a control state in which PAM executes its schedules. In Default, the base OS scheduler makes unrestricted scheduling decisions: it can assign any process (or software thread) to any CPU. PAM does not interfere, but watches the hardware performance counters monitoring various usage statistics

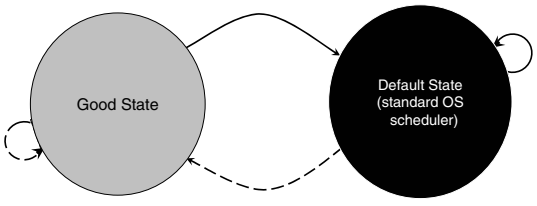

Fig. 2. PAM State Diagram.

on CPI, L1, L2 usage. If PAM predicts that better power consumption/performance may be obtained, it will produce a *plan*. A plan is basically remapping of subsets of running processes to subsets of CPUs such that the standard scheduler can schedule a process only on a CPU in the associated CPU set. If a good plan is not found, PAM continues monitoring the system in Default state. More details on detecting resource bottlenecks and producing schedules are provided at the end of this section and in Section IV. We describe here only the scheduler state machine.

If a good plan is found, we move from the Default state to the Good state while remapping process sets to CPU sets according to the plan. Remapping is implemented by the cpuset mechanism described in Section II-C. State transitions (indicated by arrows in Fig. 2) occur at discrete intervals typically at one second intervals. We chose a scheduler cycle time longer than the base kernel scheduler cycle time, but short enough so that we don't miss better scheduling opportunities by waiting too long between decisions. Very short cycles times may interfere with the kernel scheduler activity and may also increase cache misses by shuffling processes too many times between the caches (here "cycle" refers to the scheduler state transitions, not to be confused by CPU execution cycles).

Once the scheduler is in the Good state, it starts monitoring the *Goodness(t)* metric, defined later, to determine if the plan is succeeding. If *Goodness(t)* is better than that of the last Default state (which was recorded in the previous scheduling cycle), PAM stays in the Good state for N cycles. If Goodness is worse than that of Default, in M cycles PAM goes back to Default. Once the decision is made to stay in the Good state, the Goodness of the state is re-measured every M cycles. If Goodness drops below that of the Default state, PAM goes back to the Default state.

Worst case performance for the PAM meta-scheduler can be derived based on the values of N and M. Assume that the predictions are always wrong: Each time meta-scheduler transitions in to the Good state it observes less than expected goodness, and returns back to the Default state in M cycles, and then stays in Default for N cycles before making any new decisions. In this worst case scenario, system will stay in Default for $(N - M)/N$ fraction of the time on the average. By choosing $N >> M$ we can guarantee that in the worst case the system will spend most of its time in Default. Therefore, the system goodness with PAM should be no worse than $(N - M)/(N)$ of that without PAM. By choosing $N >> M$ we can also make sure the cost of reassigning processes to new CPUs (including the cost of cach misses due to the reassignments) remain negligible. Similarly, in the best case, if the predictions are always correct, scheduler will be in Good for $(N - M)/N$ fraction of the time.

Goodness(t) may be defined in multiple ways depending on whether we are optimizing for power, energy or performance. For performance, we define Goodness(t) as the $System\_Speedup$ metric defined in Eq. 1. In other words, when $System\_Speedup > 1$ scheduler stays in Good, otherwise it goes back to Default.

For the power objective, we defined goodness as $Goodness(t) = Power(t)$ such that $Power(t)$ is less than some power threshold. This is commonly referred to as *power capping*, where the objective is to ensure that the power consumption of the system does not exceed a user set threshold. Power consumption may be measured by the built-in hardware or by estimation as described in Section II-B. Note that power and energy are not to be confused. In data centers, power capping may be used to enforce some physical limits, such as a limit on the electrical feed, forcing systems to be power capped. Energy is power integrated over time. In our scenarios, we want to minimize the energy spent on completing the jobs. Thus, for the energy objective we define $Goodness(t)$ as the the energy spent per processor instruction: $(Total\_System\_Power \times cpu\_cycle\_time)/\sum IPC$.

Note also that minimizing energy does not necessarily mean minimizing power. In practice, running the processors as fast as possible (at high power) often completes the jobs faster and may lead to less energy consumption. Due to space limitations, we will present experimental results only for performance and power but not for energy.

IV. Estimating Performance of Schedules

Given a set of n processes (or n groups of processes) each running on one of the n CPUs and given the statistics collected from hardware performance counters, we compute a better performing process–to–CPU assignment and use it the next scheduling cycle (e.g. using the 2-state machine described in Section III.)

Our main contribution here is a simple algebraic model of a system of CPUs, caches, and memory and predicting performance of the system for each of the $n!$ possible process–to–CPU assignments. Our method is quite practical in the sense that (a) it works on existing x86 processors and uses existing hardware performance counters, (b) works with unmodified application binaries and for any application, (c) does not require architecture simulation or tracing or profiling or re-running applications. It should be noted that in practice due to the symmetry of real systems the number of assignments to consider is much less than $n!$.

In our experimental system each CPU had a dedicated L1, therefore the scheduling decisions did not involve L1 caches. Therefore, in the following when we refer to a cache, it means an L2 cache unless noted otherwise. In SMT systems with multiple hardware threads sharing an L1 cache, the method described here is still valid and may be used for both L1 and L2 caches without losing generality.

We first describe a sketch of the method before going in to details. We introduce a metric named "cache occupancy ratio (O)" which is a predictor of the cache footprint size of process. Hardware performance counters are used to determine the O of each process. For example, when SPEC2006 CPU benchmarks Bzip2 and Libquantum are run in parallel while sharing the same L2, we observed that on the average $O_{\text{Bzip2}} = 0.15$ and $O_{\text{Libquantum}} = 0.85$, suggesting that Libquantum is occupying most of the cache. We compute the change in O if a process shares an L2 with processes other than the current ones, and predict performance of the process as a function of O. We solve this problem for all possible process to CPU assignments ($n!$ assignments in the worst case but much less in most cases) in every scheduling cycle in order to find the highest performing schedule.

We also introduce equations estimating the cache miss ratio m_2 as a function of O, and CPI as a function of m_1, m_2, and O as a function CPI, m_1, m_2 and current cache statistics. We solve these interdependent equations iteratively to determine the System Speed of of each process-CPU assignment.

Notation used in the following sections is described first. We use a matrix notation when solving n set of equations for an n CPU system. Quantities are represented by $n \times 1$ vectors with a right arrow on top; for example $\vec{M}_2$ represents the L2 miss ratios of n CPUs. Bold face capital letters are reserved for $n \times n$ square matrices, for example $\mathbf{P}$. We also defined matrix/vector operators $*$ and $./$ and $.\wedge$ to respectively denote the **element-wise** multiplication, division and raised to the power of alpha operations, borrowing from the MATLAB notation. For $n \times m$ matrices, $\vec{A} = \vec{B} .* \vec{C}$ is defined as $a_{ij} = b_{ij} \times c_{ij}$ and $\vec{A} = \vec{B} .^{\wedge \alpha}$ is defined as $a_{ij} = b_{ij}^{\alpha}$ where $i = 0 \ldots n-1$, $j = 0 \ldots m-1$. Finally, the superscript h is affixed to quantities derived from hardware performance counters.

A. Number of Scheduling Choices

There are $n!$ ways to pair n processes (or n groups of processes) with n CPUs. We describe each pairing by an $n \times n$ permutation matrix $\mathbf{P}$ derived from interchanging rows of the identity matrix $\mathbf{I}$. Moving row i of $\mathbf{I}$ to row j implies moving quantities (such as the miss ratio) associated with a process running on CPU i to CPU j, when calculating the estimated performance of CPU $i \rightarrow j$ process re-assignment. For example, $\mathbf{P}$ shown below swaps the second and last elements of a process ID vector $\vec{\mathbf{id}}$.

$$\begin{aligned}
\vec{\mathbf{id}}' &= \mathbf{P} \times \vec{\mathbf{id}} \\
&= \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & 1 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 1 & 0 & \cdots & 0 \end{bmatrix} \times \begin{bmatrix} id_0 \\ id_1 \\ \vdots \\ id_{n-1} \end{bmatrix} \\
&= \begin{bmatrix} id_0 \\ id_{n-1} \\ \vdots \\ id_1 \end{bmatrix}
\end{aligned} \tag{2}$$

On a system with n CPUs, m caches and $n/m = k$ CPUs per cache, many of the $n!$ process assignments are performance-wise redundant due to symmetry of the hardware; for example CPUs connected to the same L2 should each yield same performance. Likewise, there should be no performance difference in running a given set of k processes on any one of the m caches (except for NUMA machines since caches are not equidistant from the distributed memory.) Therefore, the total number of process to CPU assignments to consider are

$$\Psi = \frac{\binom{n}{k} \times \binom{n-k}{k} \cdots \times \binom{k}{k}}{m!} \tag{3}$$

For the 8 CPU, 4 cache Intel system shown in Fig. 1a, the total number of permutations to consider is only

105 ($8! = 40320$). For larger n values, Eq. 3 is still a large number. In that case to reduce computation time permutations may be considered in groups of 8 CPUs at the expense of some permutations not evaluated.

We also define the $n \times n$ "cache connection matrix" $\mathbf{C}$ to describe which CPUs share which caches. $\mathbf{C}$ is a block diagonal matrix where for any two CPUs i, j sharing a cache, the elements $c_{ij} = c_{ji} = c_{ii} = c_{jj} = 1$ and all other elements are 0. For example,

$$\mathbf{C} = \begin{bmatrix} 1 & 1 & \cdots & 0 & 0 \\ 1 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 1 \\ 0 & 0 & \cdots & 1 & 1 \end{bmatrix} \quad (4)$$

B. Measuring Occupancy Ratio, Miss Ratio, and CPI

Before making the next scheduling decision we measure various cache statistics using hardware performance counters. These statistics serve as initial conditions to the calculations used for predicting performance.

We defined a metric named *cache occupancy ratio (O_i)* as a predictor of the cache footprint size of a process i:

$$L^h = L2\_LINES\_IN\_SELF + L2\_LINES\_IN\_PREFETCH \quad (5)$$

$$O_i = \frac{L_i}{\sum_{\forall \text{ process } j \in \text{ cache}} L_j} \quad (6)$$

where $L2\_LINES\_IN\_SELF$ and $L2\_LINES\_IN\_PREFETCH$ are the Intel hardware performance counters counting lines fetched from memory in to L2 [2]. The $SELF$ keyword refers to the *per core* counts for the L2 cache shared by multiple cores. Eq. 6 states that the process i fraction, O_i of the total number of lines read in to the L2 cache must be proportional to the cache footprint fraction of process i; in other words, if other processes (j) bring fewer lines in to the cache, then process i's fraction of the cache will be larger. Using matrix notation for n CPUs, Eq. 6 becomes

$$\vec{O}^h = \vec{L}^h \,./\, (\mathbf{C}\vec{L}^h) \quad (7)$$

where $\mathbf{C}$ is the cache connectivity matrix defined in Eq. 4. L1, L2 miss ratios, and CPI are calculated from Intel x86 architecture hardware performance counters [2] as

$$\begin{aligned} m_1^h &= \frac{L1D\_CACHE\_LD\_MESI}{L2\_LINES\_IN\_SELF}, \\ m_2^h &= \frac{L2\_LINES\_IN\_SELF}{LAST\_LEVEL\_CACHE\_REFS} \end{aligned} \quad (8)$$

$$CPI^h = \frac{UNHALTED\_CORE\_CYCLES}{INSTRUCTIONS\_RETIRED} \quad (9)$$

C. Estimating Performance of Scheduling Choices

In this section we develop the equations and algorithms for estimating performance of different process–to–CPU assignments. We derived a relationship between the miss ratio m_2 and the occupancy ratio O. It has been noted that a cache's miss ratio may be modeled as a function of its size S by the polynomial, the power-law curve

$$\beta S^{-\alpha} \quad (10)$$

where β, α are workload dependent constants ($\alpha > 0$) [7], [8]. The equation suggests that larger the cache size smaller the miss ratio. When $\alpha = 0.5$, it's commonly referred to as "the cache rule of thumb" which states that every doubling of the cache size reduces misses by 30%.

We hypothesized that for shared caches, miss ratio of a process as a function of its occupancy ratio O, can also be modeled by a power-law curve of the same form in Eq. 10 such that $m = \beta O^{-\alpha}$. Using two sets of this equation with different variables, we derived the relationship

$$m_2 = m_2^h \left(\frac{O^h}{O} \right)^{\alpha} \quad (11)$$

where m_2^h and O^h are known quantities derived from hardware performance counters monitored in the current scheduling cycle (Eq. 6, 8), and m_2, O, α are yet to be determined unknowns of the next scheduling cycle. Eq. 11 suggests that higher occupancy ratio results in a lower miss ratio. The constant α is a workload dependent constant indicating the sensitivity of the workload to its cache footprint size. Smaller α means that the workload's miss ratio is relatively insensitive to it's occupancy ratio. We followed the cache rule of thumb approach for non-shared caches [7], [8] and used $\alpha = 0.5$ throughout our experiments with good results as described in Section V. Although, α may be computed on-line and for each process individually by recording past values of m_2 and O; for example recording two sets of m_2 and O values in two previous scheduling cycles and substituting them in Eq. 11 would yield an α value, which then may be used in estimating the miss ratio of a process, as a function of O, in the next scheduling cycle. Using matrix notation for n CPUs, Eq. 11 becomes

$$\vec{M}_2 = (\mathbf{P}\vec{M}_2^h) \,.*\, ((\mathbf{P}\vec{O}^h) \,./\, \vec{O})^{\,.\wedge \alpha} \quad (12)$$

To estimate performance of a process, we used an approximation of CPI as a function of m_1 and m_2 in Eq. 13. The equation has the same form as the average

memory access time (AMAT) measure used in cache studies.

$$CPI = t_1 + t_2 m_1 + t_3 m_1 m_2 \quad (13)$$

In our case, t_1 represents not the L1 access time but the average CPI of a process for assuming an infinite L1 size and $m_1 = 0$. t_2 is proportional the average L2 access time and t_3 is proportional to the average memory access time. The constants t_1, t_2, t_3 may be determined either on–line or off-line on any hardware platform using statistical methods such as linear regression correlating t unknowns to the CPI^h, m_1^h, and m_2^h values obtained from hardware performance counters. For example, by recording at least three sets of CPI^h, m_1^h, m_2^h values in previous scheduling cycles and solving for t_1, t_2, t_3 as unknowns would yield Eq. 13 that can be later used as a CPI estimator as a function of m_1, m_2. On our experimental platform (Intel Xeon E5345 2.33GHz Quad-Core system (with eight cores), an IBM HS21xm blade) we monitored the CPI, m_1, m_2 metrics of two SPEC2006 CPU benchmarks Bzip2 and Libquantum and we solved the constants off-line once as

$$CPI = 0.733 + 3.43 m_1 + 110 m_1 m_2 \quad (14)$$

We hard-coded this equation in to our scheduler and used throughout for all the benchmarks used in our experiments. An on-line solution of the t constants would be more usable than Eq. 14 in general as it will work for any hardware. Using matrix notation for n CPUs, Eq. 13 becomes

$$\vec{CPI} = \vec{T_1} + (\vec{T_2} .* \vec{M_1}) + (\vec{T_3} .* \vec{M_1} .* \vec{M_2}) \quad (15)$$

We now derive the final equation estimating O as a function of CPI, m_1 and m_2. Let's assume that a process will be moved from its current CPU to another CPU and therefore it will be sharing an L2 cache with a different set of processes than the ones in the current CPU location (according to some permutation $\mathbf{P}$ as in Eq. 3). Let's also assume that the subject process will have a bigger cache footprint in the new setting; Therefore, its new O will be greater than its current O^h. Subsequently m_2 of the process should be smaller than its current miss ratio m_2^h, as Eq. 11 predicts. Thus, the rate of lines fetched in to the L2 cache should reduce by a factor of m_2/m_2^h due to reduced misses. Likewise, since the new m_2 is smaller, the new CPI of the process should be smaller than the current CPI^h, as Eq. 13 predicts. Since a smaller CPI implies a faster process, it should make more memory references by a factor of CPI^h/CPI per unit time. Overall, number of lines fetched in to the cache per unit time becomes

$$L = L^h \times \frac{m_2}{m_2^h} \times \frac{CPI^h}{CPI} \quad (16)$$

where L^h, m_2^h, and CPI^h are measured values from hardware performance counters (Eq. 5,8,9) and L, m_2, and CPI are yet to be determined unknowns. Using matrix notation for n CPUs, Eq. 16 becomes

$$\vec{L} = \vec{L^h} * (\vec{M_2} ./ (\mathbf{P}\vec{M_2^h})) * ((\mathbf{P}\vec{CPI}^h) ./ \vec{CPI}) \quad (17)$$

Now, we have a complete set of equations Eqs. 7, 12, 15, and 17 for determining all the unknowns $\vec{L}$, $\vec{O}$, $\vec{M_2}$, and $\vec{CPI}$, iteratively:

Algorithm1:

Inputs: (1) A permutation matrix $\mathbf{P}$ specifying a what-if scenario of shuffling n processes among n CPUs, (2) Hardware performance counter measurements collected from n CPUs, $\vec{L^h}$, $\vec{M_1^h}$, $\vec{M_2^h}$, $\vec{CPI}^h$ (Eqs. 5, 8, 9),

Output: Total system speedup as defined by Eq. 1 and $\vec{L}$, $\vec{O}$, $\vec{M_2}$, and $\vec{CPI}$.

1) Initialize the n elements of $\vec{O}$ to any initial constant, for example $O_i = 1/n$, $(i = 0 \ldots n-1)$
2) Compute the miss ratios $\vec{M_2}$ using Eq. 12
3) Compute the CPU rates $\vec{CPI}$ using Eq. 15 and $\vec{M_2}$ computed in the previous step
4) Compute the rate of lines fetched $\vec{L}$ in to the caches using Eq. 17, and $\vec{M_2}$ and $\vec{CPI}$ computed in the previous two steps
5) Compute the new occupancy ratio as $\vec{O}^{\text{new}} = \vec{L} ./ \mathbf{C}\vec{L}$ (similar to Eq. 7) using $\vec{L}$ computed in the previous step
6) Check if iterations converged: **if** $\|\vec{O}^{\text{new}} - \vec{O}\| > 0.001$, **then** $\vec{O} \leftarrow \vec{O}^{\text{new}}$ and go to step 2 and repeat using the new $\vec{O}$ values **else** $\vec{O} \leftarrow \vec{O}^{\text{new}}$ and go to next step **endif**
7) Compute the total system speedup using $\vec{CPI}$ computed in step 3 and using the speedup definition of Eq. 1 and return

We observed that the algorithm terminates typically in 5–6 iterations in practice. Algorithm1 estimates the performance for a single permutation. Recall from Eq. 3 that there are many process–to–CPU assignments to evaluate the performance of, which is performed by the following:

Algorithm2:

Inputs: Hardware performance counter measurements collected from n CPUs, $\vec{L^h}, \vec{M_1^h}, \vec{M_2^h}, \vec{CPI}^h$

Output: The permutation matrix $\mathbf{P}$ specifying the best performing process–to–CPU assignment which will be used for the next scheduling cycle

1) Initialize: $min\_moves \leftarrow \infty$, $max\_perf \leftarrow 1$, $\mathbf{PP} \leftarrow \mathbf{I}$ (identity matrix)
2) Generate all non-redundant permutations $\mathbf{\Phi} = \{\mathbf{P}_1 \ldots \mathbf{P}_\kappa\}$ (see Section IV-A)
3) **forall** $\mathbf{P} \in \Phi$ **do**
   a) Compute performance of $\mathbf{P}$ as $perf \leftarrow \text{Algorithm1}(\mathbf{P}, \vec{L}^h, \vec{M}_1^h, \vec{M}_2^h, \vec{CPI}^h)$
   b) Compute number of process movements $nmoves$ due to $\mathbf{P}$
   c) **if** $perf > max\_perf + \delta$ **or** $(perf > max\_perf - \delta$ **and** $nmoves < min\_moves)$ **then**
      $\mathbf{PP} \leftarrow \mathbf{P}$
      $min\_moves \leftarrow nmoves$
      $max\_perf \leftarrow perf$
      endif
   endfor
4) Return $\mathbf{PP}$

The logic behind Steps 3.b and 3.c is to make sure that a permutation $\mathbf{P}$ with insignificant performance gain is not selected as determined by the constant δ. Likewise a lower performing permutation than the previously chosen one is accepted if the permutation has fewer processes moving between the CPUs (e.g. identity matrix has 0 moves). In our experiments we used $\delta = 0.1\%$.

V. Performance Evaluation

In order to evaluate our meta-scheduler, we have built a prototype. This prototype works with Linux systems and Intel processors. In addition to hardware counters, power counters available on IBM blades were used for measuring power consumption. In order to evaluate the impact of PAM we used synthetic programs and benchmarks such as SPEC CPU 2006. In this section, we first present our test-bed and then present the results.

A. Testbed

We used an 8-core IBM blade server with Intel Xeon E5345 processors. In this system there are four chips (in two packages). In each chip, there are two cores which share an L2 cache. The cache organization of this system is the same as the one shown in is shown in Figure 1.a. The CPU and L2 speed is 2.33 GHz and each pair of cores share a 4 MB L2 cache. The server is used with 16 GB of memory with fully-buffered DIMMs. This server runs Intel Speedstep, in DYNAMIC mode with CPUs idling at 2.0 GHz.

B. Base Results

We used the SPEC CPU 2006 benchmark suite and synthetic programs to study the impact of PAM. Obviously, when there is only one program running on the system, there is not much one can do with respect to tuning the performance. On the other hand, when multiple applications are running on a system, chances are that these applications have different characteristics and behaviors and therefore tuning the system using PAM can have a significant impact.

In particular, we used SPEC CINT2006 benchmarks to study a wide range of environment with respect to number of software threads and number of benchmarks running at the same time. We first validated Equation 11. We measured the occupancy and miss ratio of each benchmark in CINT2006 while running with all the benchmarks in this suite one at a time. Then, in order to get an idea about the interaction of benchmarks in this suite, we performed a test when the performance of a given benchmarks was measured in the presence of a disturber benchmark on the same CPU package (i.e., with shared cache). The results are shown in Figure 3. It can be seen that benchmarks list on the right side of the x axis have a bigger impact on other benchmarks.

We ran several tests with various combinations of benchmarks and number of software threads (processes). Figure 4 shows the results from a set of tests in which four Libq threads were run with four threads of every other benchmark in the CINT2006 suite. Results are normalized with respect to the execution time in the standard system without PAM. For each set of bars, Libq and the other benchmark were ran together at least 3 times. In all cases the numbers reported reflect the execution time of benchmarks when 8 processes (4 Libq processes and 4 processes from another benchmark) were running. It should be noted that in all experiments the execution time was recorded as reported by the SPEC benchmark and reduction (or increase) in execution times is calculated based on these execution times.

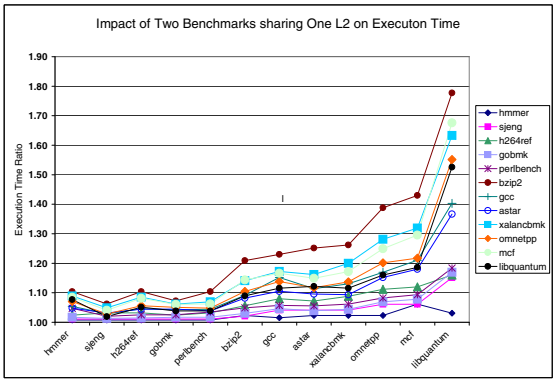

Fig. 3. Interaction between SPEC CINT2006 benchmarks.

It can be seen in Figure 4 that when Libq and Mcf

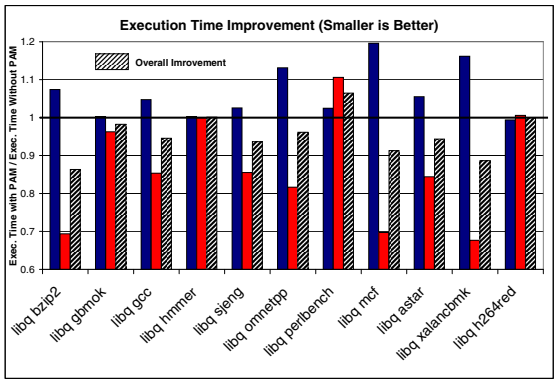

Fig. 4. Libq and other CINT2006 benchmarks. In each group of bars, the left bar corresponds to Libq and the middle bar corresponds to other benchmarks. The overall improvement is shown by the right bar in each group.

benchmarks are run together even though the overall performance is improved, the performance of Libq itself degrades significantly. As mentioned in Section IV, the maximum degradation of individual benchmarks can be specified in PAM. Figure 5 shows the impact of changing the maximum degradation allowed (in percentage) for Libq-Mcf run. By default, this is set to 10% meaning the performance of each individual benchmark can degrade to as little as 10% of its original performance. This is essentially set to such a low value to remove any practical limit on individual benchmark performances. It can be observed that when this limit is set to 95%, Libq degradation is reduced substantially and overall performance improvement decreases.

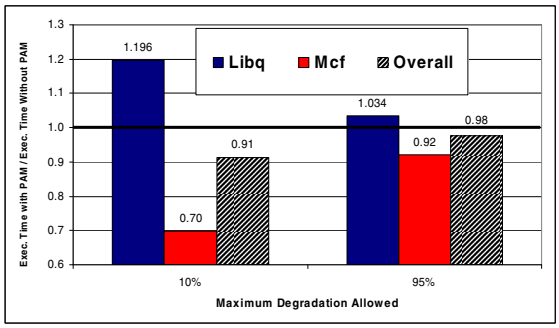

Fig. 5. Limiting degradation of individual programs.

C. CPU Occupation

In order to illustrate how different processes get scheduled with and without PAM, we periodically recorded the CPU id of processes being run. How Libq and Xalancbmk processes are scheduled in two runs without and with PAM is illustrated in Figure 6. It can be seen that without PAM, once Libq and Xalancbmk are scheduled and assigned to a CPU, they are rarely reassigned to a different CPU. It can be seen in the left-side graph of Figure 6 that when PAM is not used, on each chip there is one Libq and one Xalancbmk processes. This means that in all chips Libq and Xalancbmk processes share the L2 cache. On the other hand, with PAM (right-side graph of Figure 6) there are more changes in process-CPU assignments. More importantly, PAM makes sure Xalancbmk processes are scheduled on the same chips and Libq processes are scheduled on other chips. From Figure 4, it can be observed that Libq suffers a bit while the performance of Xalancbmk improves significantly resulting an overall improvement.

D. Miss Ratio

To get a better understanding of the impact of PAM, we also recorded L1/L2 miss rates among other hardware data. Figure 7 shows the L2 miss ratio of the CPU under three different conditions. Please note the logarithmic scale. When PAM is used, two Xalancbmk processes run on a chip. The L2 miss ratio of one such CPU is shown. Furthermore, the L2 miss ratio of Xalancbmk when two such processes are on one chip is shown as well. Furthermore, the L2 miss ratio of a CPU running Xalancbmk when the other CPU on the same chip is running Libq is shown. It can be seen that after the initial stage of the run which causes significant miss rates, the miss rate when Xalancbmk and Libq share an L2 remains significantly higher. This condition which occurs often with standard scheduling leads to a bad performance. On the other hand when Xalancbmk processes are sharing L2, the miss ratio is much less. The initial high miss ratio is also shorter in such a condition. When PAM is used, processes are scheduled such that for the most part we will be having this later condition and therefore a better performance.

E. Multiple Benchmarks

We have also run various other combination of benchmarks and number of software threads in our experiments. The results of experiments with running 3 benchmarks are shown here. In Figures 8 and 9 we show the result of some representative sets of experiments. In Figure 8, first and third benchmarks have three software threads while the second benchmark runs with two threads. In Figure 9, all benchmarks run with three threads each. In all experiments the reported results are obtained while all three benchmarks were running. It

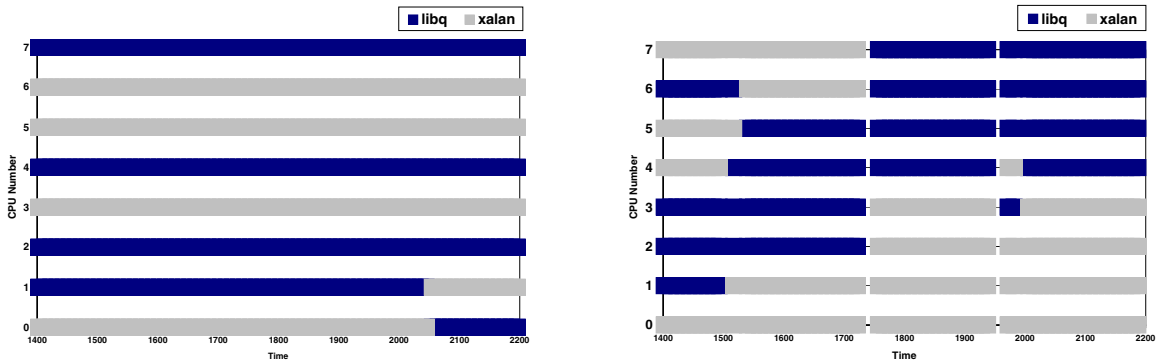

Fig. 6. Core assignments with PAM. It can be observed that without PAM (left), running software threads are randomly assigned to hardware cores. When PAM is used (right), it is immediately detected that running certain software thread on the same CPU is beneficial. In this example, PAM detects that running Libq and Xalancbmk threads such that they do not share L2 caches results in a better performance.

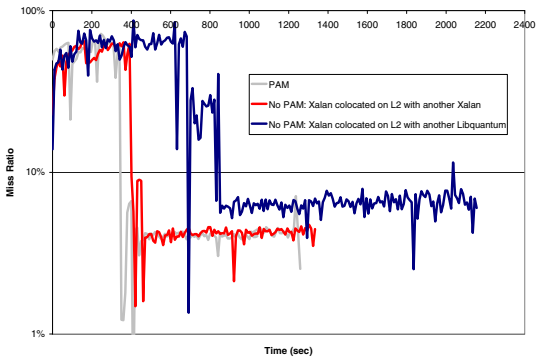

Fig. 7. Miss ratio for the Xalancbmk benchmark.

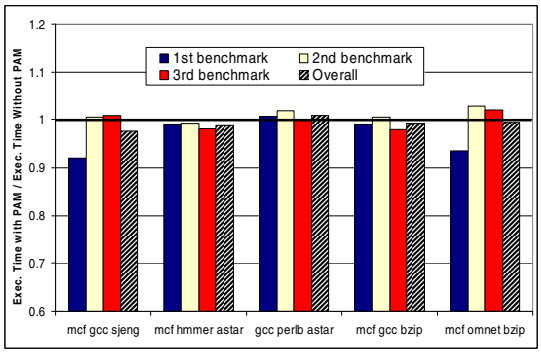

Fig. 8. Multi-benchmark experiments with 3 benchmarks running with 3, 2, and 3 threads respectively.

can be observed that in most cases using PAM leads to a better performance. It is to be noted that even when the number of software threads are higher than that of CPUs (or hardware threads) using PAM is beneficial. In these cases, when multiple processes run on a single core, they are treated as one group and are moved together.

F. Floating Point

In addition to integer benchmarks, we have performed several tests with floating point benchmarks (i.e., SPEC CFP2006). A subset of these results are shown in Figure 10.

G. Power

In order to evaluate the effectiveness of PAM with respect to capping the power, we set the power cap to different values and observed how PAM reduces and/or increases the number of CPUs being used. For a certain power cap, the performance can be optimized by PAM by using the same algorithms described earlier only on a limited number of CPUs. Figure 11 shows the number of

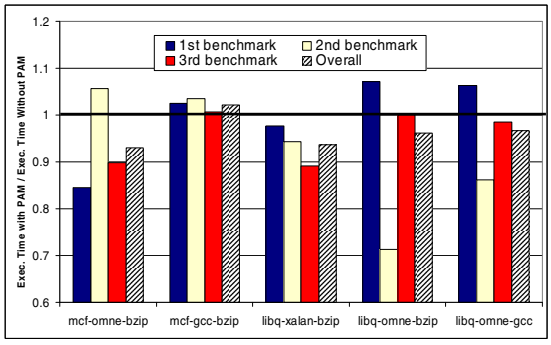

Fig. 9. Multi-benchmark experiments with 3 benchmarks running with 3 threads each.

CPU used and the amount power used by the server as recorded in hardware power counters. For this particular experiment we ran the whole SPEC CINT2006 and set the power cap to 260 watts. It can be seen that in most cases four or five CPUs are used to make sure no

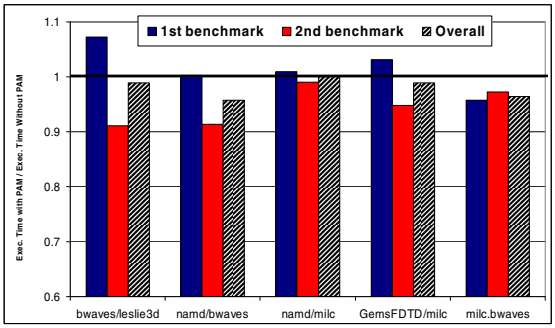

Fig. 10. SPEC CFP2006 benchmarks.

more power is used. It can be observed that between benchmark runs when no benchmark is running, the number of used CPUs increase to eight momentarily and as the next benchmark starts running the number of CPUs are decreased.

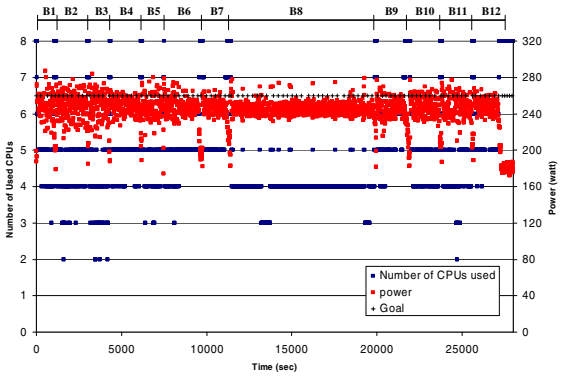

Fig. 11. Power consumption for CINT2006 benchmarks.

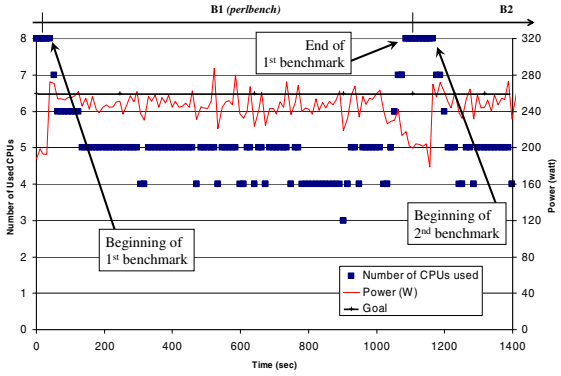

Fig. 12. Power consumption blow out for the Perlbench benchmark.

Figure 12 shows the same result for a smaller time period. Data shown in this figure is for the Perlbench benchmark. It can be seen that with four or five CPUs the power is capped at around 260 watts. It should be noted that the Intel processors used in this server are of the energy saving type and the range of frequency/voltage change is very narrow (2.0 GHz to 2.3 GHz). On the other hand on our servers with AMD processors there is a wider range of valid frequency/voltage that one can use to reduce the power consumption. PAM is also capable of applying its scheduling technique to improve the performance as much as possible while the power consumption is capped.

VI. Related Work

A memory-aware scheduling scheme is presented in [9]. In this work, it is proposed to use a new set of hardware counters such that the miss ratio of a given process with a certain cache size can be predicted. This information is then used to provide for using a better schedule to minimize the overall miss rate in the system. Unlike this work, PAM does not require any new hardware counters and rely on the counters already available on modern CPUs.

A dynamic cache partitioning method for minimizing the overall miss rate and improving IPC is presented in [10]. This method which aims to replace the LRU replacement policy currently used in systems with a new replacement policy. A group of cache policies for chip multiprocessors are discussed in [11]. A hardware cache quota system that can be used by OS to use different policies for different applications in order to improve the overall performance in chip multiprocessors is proposed in [12]. The approach taken in our work does not require any changes in replacement policies or any other components of the system.

Mechanisms for improving the performance by using better scheduling in simultaneous multithreading machines are discussed in [13]. These methods are executed in two phases: first a sample phase which collects information about various possible schedules, and second a symbiosis phase in which the information collected in phase one is used to predict which schedule will provide the best performance.

Improvement in the schedulers for multithreaded chip multiprocessors by balancing the use of shared L2 caches is discussed in [14]. The proposed scheme relies on *balance-set* scheduling where groups of processes whose combined working set fits in the cache. In order to model the cache behavior accurately, a cache model for multithreaded applications is developed which requires the memory reuse pattern for accurately estimating the

cache miss rate. As mentioned in this paper, using such an approach in a real system is expensive and impractical.

A new operating system scheduling algorithm which provide performance isolation on chip multiprocessors are discussed in [15]. In this algorithm, if a thread is affected by other threads in a negative way, the operating systems increases that thread's CPU time slice such that its overall performance does not suffer. Operating system scheduling for heterogeneous multi-core systems are discussed in [16], [17]

Issues involving an accurate metric for measuring the performance on multiprocessors have been discussed in [3], [18]. It can be easily observed that in multiprocessors use of a metric such as IPC can be misleading when various architectural changes of the system is being investigated. This issue becomes even more complicated as multiple multi-thread processes with different characteristics run on such systems. We mentioned earlier that in such cases, a simple average may not be accurate and weighted average, or geometric means of improvement rates can be used.

VII. Conclusions and Future Work

In this paper we showed how information obtained from hardware performance counters and power counters can be used to monitor the state of the system. we further showed how new scheduling plans can be used to improve the system. we developed a model for estimating the impact of new schedules and showed they can be of significant value for a large set of benchmarks.

We are extending our work along several directions. we are evaluating and comparing several other performance metrics. We are extending our test-bed to include non-Intel processors and servers with different number of cores. We are also considering the impact of memory affinity on the performance.

References

[1] "Perfmon2: The Hardware-based Performance Monitoring Interface for Linux," 2008, http://perfmon2.sourceforge.net/.

[2] Intel Corporation, "Intel 64 and ia-32 architectures software developers manual, volume 3b: System programming guide, part 2," May 2007.

[3] A. R. Alameldeen and D. A. Wood, "IPC Considered Harmful for Multiprocessor Workloads," *IEEE Micro*, vol. 26, no. 4, pp. 8–17, 2006.

[4] IBM Corporation, "IBM PowerExecutive," 2008, http://www.ibm.com/systems/management/director/extensions/powerexec.html.

[5] C. Lefurgy, X. Wang, and M. Ware, "Server-Level Power Control," in *Proceedings of the Fourth International Conference on Autonomic Computing (ICAC'07)*, 2007.

[6] "CPUSET Filesystem," 2008, http://www.bullopensource.org/cpuset/csfs.html.

[7] B. L. Jacob, P. M. Chen, S. R. Silverman, and T. N. Mudge, "An Analytical Model for Designing Memory Hierarchies," *IEEE Transactions on Computers*, vol. 45, no. 10, pp. 1180–1194, 1996. [Online]. Available: citeseer.ist.psu.edu/article/jacob96analytical.html

[8] A. J. Smith, "Cache Memories," *Computing Surveys*, vol. 14, no. 3, pp. 473–530, 1982.

[9] G. E. Suh, S. Devadas, and L. Rudolph, "A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning," in *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA)*, 2002, pp. 117–. [Online]. Available: citeseer.ist.psu.edu/suh02new.html

[10] G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic Partitioning of Shared Cache Memory," *The Journal of Supercomputing*, vol. 28, no. 1, 2004.

[11] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni, "Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches As a Shared Resource," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006.

[12] N. Rafique, W.-T. Lim, and M. Thottethodi, "Architectural Support for Operating System-driven CMP Cache Management," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006.

[13] A. Snavely and D. M. Tullsen, "Symbiotic Jobs Scheduling for a Simultaneous Multithreaded Processor," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000, pp. 234–244.

[14] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum, " Performance Of Multithreaded Chip Multiprocessors And Implications For Operating System Design," in *Proceedings of the USENIX 2005 Annual Technical Conference*, 2005.

[15] A. Fedorova, M. Seltzer, and M. D. Smith, "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler," in *Proceedings of the Sixteenth International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2007.

[16] A. Fedorova, D. Vengerov, and D. Doucette, "Operating System Scheduling on Heterogeneous Core Systems," in *Proceedings of the First Workshop on Operating System Support for Heterogeneous Multicore Architectures, in conjunction with PACT*, 2007.

[17] D. Shelepov and A. Fedorova, "Scheduling on Heterogeneous Multicore Processors Using Architectural Signatures," in *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture, in conjunction with ISCA*, 2008.

[18] K. M. Lepak, H. W. Cain, and M. H. Lipasti, "Redeeming IPC as a Performance Metric for Multithreaded Programs," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2003.