CPU Scheduling for Power/Energy Management on Multicore Processors Using Cache Miss and Context Switch Data

Ajoy K. Datta and Rajesh Patel

Abstract—Power and energy have become increasingly important concerns in the design and implementation of today's multicore/manycore chips. In this paper, we present two priority-based CPU scheduling algorithms, Algorithm *Cache Miss Priority CPU Scheduler* ($\mathcal{CM}$*-PCS*) and Algorithm *Context Switch Priority CPU Scheduler* ($\mathcal{CS}$*-PCS*), which take advantage of often ignored dynamic performance data, in order to reduce power consumption by over 20 percent with a significant increase in performance. Our algorithms utilize Linux cpusets and cores operating at different fixed frequencies. Many other techniques, including dynamic frequency scaling, can lower a core's frequency during the execution of a non-CPU intensive task, thus lowering performance. Our algorithms match processes to cores better suited to execute those processes in an effort to lower the average completion time of all processes in an entire task, thus improving performance. They also consider a process's cache miss/cache reference ratio, number of context switches and CPU migrations, and system load. Finally, our algorithms use dynamic process priorities as scheduling criteria. We have tested our algorithms using a real AMD Opteron 6134 multicore chip and measured results directly using the "KillAWatt" meter, which samples power periodically during execution. Our results show not only a power (energy/execution time) savings of 39 watts (21.43 percent) and 38 watts (20.88 percent), but also a significant improvement in the performance, performance per watt, and execution time · watt (energy) for a task consisting of 24 concurrently executing benchmarks, when compared to the default Linux scheduler and CPU frequency scaling governor.

Index Terms—Multicore processor, manycore processor, CPU intensive task, global power budget, dynamic process priority, heterogeneous architecture, CPU frequency scaling governor, cpuset, hardware performance counter, dynamic CPU frequency scaling, memory performance saturation, Linux nice value, cache coherence, CPU migration

1 Introduction

Recent advances in processor production have led to not only an increase in the number of cores on a single chip, but also a concomitant increase in power consumption and heat dissipation associated with these high performance systems. For example, a 2.8 GHz Intel "Northwood" Pentium 4 chip consumes around 80 W, while a "Prescott" Pentium 4 chip of the same speed consumes around 100 W [1]. This increased power consumption creates an associated increase in heat dissipation, which leads to higher costs for air conditioning, thermal packaging, fans, and electricity. Increased heat and temperature of hardware, such as the CPU, can also lead to decreased longevity and greater incidence of failure of these components. Also, the integration and design of a multicore chip is more difficult to manage, than a lower-density, single-chip design due to thermal constraints [2]. Finally, power savings and decreased heat production can be crucial in notebook computers, where battery life is a constraint and cooling of hardware components is difficult, and in server farms, where even a slight decrease in the power consumption of each server can cause a significant savings for the entire system.

Scaling a CPU's frequency can dramatically affect its power consumption. In fact, the power dissipation at the output pins of a core is directly proportional to its frequency and is governed by the equation

$$P = \frac{1}{2}CV^2 f$$

where C is capacitance, V is voltage, and f is the effective bus frequency.

Dynamic voltage scaling (DVS) is another CPU power management technique which involves dynamically decreasing the voltage used by the CPU, depending upon circumstances, in an effort to conserve power. Despite the quadratic relationship between voltage and power consumption, DVS causes an increase in circuit delay, thus creating a performance tradeoff.

CPU frequency scaling may be used to create a homogeneous architecture with cores having the same instruction set, but that operate at different frequencies. The class of architectures used in this research is such an

• *The authors are with the Department of Computer Science, University of Nevada at Las Vegas, Las Vegas, NV 89154 USA. E-mail: Ajoy.Datta@unlv.edu; patel2@unlv.nevada.edu.*

Manuscript received 22 Apr. 2012; revised 21 Aug. 2012; accepted 9 May 2013. Date of publication 22 May 2013; date of current version 21 Mar. 2014. Recommended for acceptance by J. Flich.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2013.148

architecture, in which the processors have the same instruction set and microarchitecture but that operate at different frequencies, adopting dynamic frequency scaling (DFS) to reduce power consumption. Any reference to a multicore processor or system used for this research shall imply such an architecture.

Heterogeneous architectures may achieve a higher performance per watt than comparable homogeneous systems [3], [4] due to the ability of each application to run on a core that best suits its architectural properties. Even though architectures with cores operating at different frequencies may still be considered homogeneous, such architectures may contain cores specialized for certain tasks. For example, threads conducting CPU intensive work may execute upon high frequency cores, while threads conducting memory intensive work may execute upon low frequency cores, which consume significantly less power.

Due to the slow speed of cache and memory access relative to processor speed, processes (whose performance is limited by available memory bandwidth [5], [6]) may obtain limited benefit from increasing processor frequency. Thus when executing a task containing both CPU intensive and memory intensive processes, it may be beneficial to move such memory intensive processes to lower frequency cores, while allowing less memory intensive, more CPU intensive processes to execute upon higher frequency cores.

Context switching less CPU intensive processes, that would benefit less from executing upon higher frequency cores, to lower frequency cores and allowing CPU intensive processes to execute upon higher frequency cores may minimize performance loss while providing significant power savings. Doing so while utilizing process priority as a scheduling criterion is also beneficial.

Since a process which is context switched too often may not find valid data in its new core's cache after being migrated to a new CPU, this process will have a tendency to generate many cache misses. This overhead associated with cache coherence, and with context switching itself, can degrade the performance of a multicore processor system. Thus it may be advantageous to implement a different strategy in this scenario. This research's goal is to identify and quantify these advantages.

1.1 Contributions

Like some recent work using heterogeneous processors [6], [7], [8], [9], [10], this research uses processors that have the same instruction set architecture (ISA) and microarchitecture but different implementation characteristics, such as operating frequency. In contrast to these works, however, our paper offers the following contributions.

We present two CPU scheduling algorithms, Algorithms $\mathcal{CM}$-PCS and $\mathcal{CS}$-PCS, that lower the global power budget in a multicore (or manycore) processor system while creating a performance gain (and an improvement in performance-energy metrics) given a task containing a set of processes to be executed on this system. These algorithms utilize hardware partitions composed of cpusets containing varying numbers of cores. The cores are identical, with the exception that cores belonging to the same cpuset operate at the same frequency, while cores belonging to different cpusets operate at different frequencies.

Hence, in our evaluation, we try to emulate the most common commercially available multicore chips. Also, since powering down cores to save energy can cause large latencies while waiting for these cores to power up again, we evaluate a theoretically more efficient scenario involving the use of different fixed frequency cores.

In this research, we address server platforms utilizing multicore CPUs that can be frequency scaled, such as the AMD Opteron or Intel Nehalem processors, running a typical server load, which may include a heavily loaded system running a task consisting of up to twenty-four concurrently executing benchmarks. Also, in order to exclude all other variables, we wanted to assess the potential power savings and performance improvements achieved by CPU scheduling and multicore CPU frequency scaling alone, assuming all other platform resources upon various server systems are equal, including CPUs or hard drives. Hence, in our survey, we evaluate hardware components that are identical or similar, which may be the case in a typical server rack.

1.2 Outline of Paper

In the supplementary file to this paper which is available in the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/148, Section 1, we explain the experimental setup and methodology used to generate a multicore architecture, with cores operating at different frequencies, and discuss how this methodology was used to implement and test our CPU scheduling algorithms. A discussion of related work is included in Section 2 of the supplementary file available online. In Sections 2 and 3 of this main file, we present our two CPU scheduling algorithms, along with a formal description of their data structures and of the algorithms themselves. We present a preliminary evaluation of our algorithms and an analysis of our experimental results in Section 4 of this main file. Finally, in Section 5 of this main file, we summarize our work and propose ideas to extend this research.

2 Feedback-Driven Priority-Based CPU Scheduling Algorithm 1

In this section we present an application-driven feedback mediated CPU scheduling algorithm that considers the ratio between the total number of cache misses and the total number of cache references generated at runtime. It also utilizes static or dynamic process priorities, and is referred to as Algorithm *Cache Miss Priority CPU Scheduler* ($\mathcal{CM}$-PCS).

2.1 Variables and Constants Used in Algorithm CM-PCS

P_i :: a single or multithreaded process;

T :: a task comprised of one or more processes (P_i) to be scheduled and executed by a multicore or manycore chip;

num_cpus :: number of CPUs (cores) on multicore or manycore chip;

$core_i$ :: a single core (CPU) on a multicore or manycore chip;
$cpuset_i$ :: a Linux cpuset, comprised of one or more homogeneous $core_i$'s,

$$(core_{i_1}, core_{i_2}, \ldots, core_{i_n})$$

$$\begin{aligned} & core_{i_1} \text{ and } core_{i_2} \text{ belonging to } cpuset_i \\ & \Longrightarrow \text{operating frequency of } core_{i_1} \\ & \qquad = \text{operating frequency of } core_{i_2} \end{aligned}$$

$$\begin{aligned} & core_{i_1} \text{ and } core_{j_1} \text{ belonging to } cpuset_i \text{ and } cpuset_j \\ & \Longrightarrow \text{operating frequency of } core_{i_1} \\ & \qquad \neq \text{ operating frequency of } core_{j_1} \end{aligned}$$

$core\_freq_i$ :: operating frequency of any $core_i$ in $cpuset_i$ (MHz);
$ready\_queue_i$ :: ready queue of any core in $cpuset_i$;
$avg\_ready\_queue\_threads_i$ :: average number of threads in $ready\_queue_i$;
$load\_estimate_i$ :: estimate of the system load of $cpuset_i$;

```
if ($avg\_ready\_queue\_threads_i \geq 1$)
    $load\_estimate_i$
        = $avg\_ready\_queue\_threads_i$
          * $(1/core\_freq_i)$;
    else
      $load\_estimate_i$
          =$1/core\_freq_i$;
```

$num\_threads_i$ :: number of threads in process P_i;
IPC_i :: instructions per cycle for process P_i;
$IPC_i = \sum_{j=0}^{num\_threads_i} IPC_j / num\_threads_i$, where j is a thread in P_i;
$system\_time_i$ :: time spent by P_i while executing at the system level (kernel);
$user\_time_i$ :: time spent by P_i while executing at the user level (application);
$real\_time_i$ :: $system\_time_i + user\_time_i$;
$fraction\_wait\_time_i$ :: fraction of total execution time spent by process P_i, waiting in the ready queues and I/O queues;
$fraction\_wait\_time_i$
$=\sum_{j=0}^{num\_threads_i} ((real\_time_j - (system\_time_j + user\_time_j))/real\_time_j)$
$/num\_threads_i$
where j is a thread in P_i;
$performance\_index_i$ :: a Boolean value calculated for an executing process P_i from IPC_i and $fraction\_wait\_time_i$;
$Performance\_index_i$ = true iff $IPC_i < 1.7$ and $fraction\_wait\_time_i > 0.5$;
$cache\_miss_i$ :: the number of cache misses generated by process P_i;
$cache\_reference_i$ :: the number of cache references generated by process P_i;
$cache\_miss\_index_i$ :: a Boolean value calculated for an executing process P_i that uses the ratio between $cache\_miss_i$ and $cache\_reference_i$;
$cache\_miss\_index_i$ = true iff
$cache\_miss_i / cache\_reference_i > 0.0014$;

$CPU\_intensity_i$ :: measure of the CPU intensity of process P_i; average CPU utilization of all cores upon which P_i is executing;
$nice_i$ :: the priority of process P_i; may be a value between −20 and 19; a lower value indicates a higher priority;
$CPU\_intensity\_priority\_index_i$ :: an index that takes into account the CPU intensity and priority of process P_i
$CPU\_intensity\_priority\_index_i$
$=(0.5 * CPU\_intensity_i) + (0.5 * \frac{1}{nice_i})$;
$load\_average$ :: average system load over a period of time; number of processes using or waiting for all CPUs (all $core_i$'s);

2.2 Algorithm CM-PCS

Algorithm 1: *Cache Miss Priority CPU Scheduler* ($\mathcal{CM}$-$\mathcal{PCS}$)

```
1: Start with a set $S$ of processes $P_i$ in a given task $T$:

2: for all $P_i$ in $T$ (concurrently){

3:   //initially schedule a process
4:   calculate $load\_estimate_i$ for all cpusets;

5:   assign $P_i$ to cpuset with least value of
6:    $load\_estimate_i$;

7:   //if non-executing process is in wait queue
8:     too long, context switch process
9:   if($fraction\_wait\_time_i \geq 0.7$ and $nice_i < -1$){
10:     recalculate $load\_estimate_i$ for all cpusets
11:     and assign $P_i$ to cpuset with least
12:     value of $load\_estimate_i$;
13:     }
14:     else if($fraction\_wait\_time_i > 0.9$){
15:         recalculate $load\_estimate_i$ for all cpusets
16:         and assign $P_i$ to cpuset with least value
17:         of $load\_estimate_i$;
18:     }

19:   do{
20:     execute $P_i$;

21:     calculate $CPU\_intensity_i$ for $P_i$;

22:     calculate $IPC_i$ and $fraction\_wait\_time_i$
23:     for $P_i$;

24:     compute $performance\_index_i$ of $P_i$ using
25:      $IPC_i$ and $fraction\_wait\_time_i$;

26:     **compute $cache\_miss\_index_i$ of $P_i$ using**
27:      **$cache\_miss_i$ and $cache\_reference_i$;**

28:     measure $load\_average$;
29:     if($CPU\_intensity_i > 50\%$ and $nice_i < 9$
30:         and $performance\_index_i$ == true
31:         and $cache\_miss\_index_i$ == false){
32:           recalculate $load\_estimate_i$ for all
33:           cpusets and context switch $P_i$ to
34:           cpuset with least value of $load\_estimate_i$;
35:         }
36:         else if($CPU\_intensity_i < 50\%$ and $nice_i > -1$
37:                 and $load\_average > num\_cpus$){
38:                   recalculate $load\_estimate_i$ for all
39:                   cpusets and context switch $P_i$ to
40:                   cpuset with greatest value of
41:                    $load\_estimate_i$;
42:               }
43:             else if($CPU\_intensity_i < 50\%$
44:                 and $nice_i \leq -1$
45:                 and $load\_average > 2 * num\_cpus$){
46:                   recalculate $load\_estimate_i$ for all
47:                   cpusets and context switch $P_i$ to
48:                   cpuset with greatest value of
49:                    $load\_estimate_i$;
50:               }

51: else if($CPU\_intensity_i < 50\%$ and $nice_i \leq -11$
52:       and $load\_average \leq num\_cpus$
53:       and $performance\_index_i$ == true
54:       and $cache\_miss\_index_i$ == false){
55:         recalculate $load\_estimate_i$ for all cpusets
56:         and context switch $P_i$ to cpuset with least
57:         value of $load\_estimate_i$;
58:   }
59:   else if($CPU\_intensity_i < 50\%$ and $nice_i > -11$
60:           and $load\_average < 0.15 * num\_cpus$
61:           and $performance\_index_i$ == true
62:           and $cache\_miss\_index_i$ == false){
63:             recalculate $load\_estimate_i$ for all cpusets
64:             and context switch $P_i$ to cpuset with least
65:             value of $load\_estimate_i$;
66:         }
67: //if a ready queue is empty, then assign a process
68:   to it
69:   if(a ready queue is empty and $P_i$ is process with
70:     highest value of $CPU\_intensity\_priority\_index_i$
71:     of all executing processes)
72:     {
73:         context switch $P_i$ to cpuset with the empty
74:         ready queue;
75:     }
76: }until $P_i$ terminates;

77:}
```

2.3 Description of Algorithm CM-PCS

A description of Algorithm $\mathcal{CM}$-PCS is as follows:

1. All processes are assigned a nice value, which signifies the priority of that process. This nice value may be between -20 and 19, with a lower value indicating a higher priority.
2. Processes are assigned to the ready queues of CPUs in a cpuset using a "load estimate" defined by the following equation: (Lines 4-6)
   If (avg. # of threads in ready queues of a cpuset $\geq$ 1)
   a) $load\_estimate =$ avg. # of threads in that cpuset's ready queues $*$ (1/frequency of any core in that cpuset)

(In case of a tie, assign process to cpuset with least average number of threads in ready queues.)
Else

a) $load\_estimate$ = 1/frequency of any core in that cpuset

Note: In Linux, there is no direct measurement of the number of processes in the ready queue of a single core (CPU) on a multicore chip. However, if processes are allocated to certain cpusets, an estimate of the average number of threads in the ready queue of a particular cpuset's core can be obtained by calculating ((run queue size of all ready queues combined/ number of CPUs where CPU utilization = 100%) x average CPU utilization of CPU in that cpuset).
Processes assigned to cores with a lower value of $load\_estimate$ are moved to cores operating at a higher frequency with fewer processes in their ready queues, while processes assigned to cores with a higher value of $load\_estimate$ are moved to cores operating at a lower frequency with a greater number of processes in their ready queues. Thus $load\_estimate$ is useful in a context where it is advantageous to move, for example, a CPU intensive process with slower performance to a higher frequency core or a non-CPU intensive process, or a process with a higher performance (relative to other processes in a task), to a lower frequency core.

3. All newly-arriving processes are assigned to the cpuset with the least value of $load\_estimate_i$. If possible, all newly-arriving processes are assigned to the cpusets with CPUs operating at higher frequencies and with fewer processes in their ready queues. (Lines 5-6)
4. As a process P_i executes, a "performance index" is calculated from the IPC (instructions per cycle) and fraction wait time of P_i. This calculation is made asynchronously with respect to other concurrently executing processes. (Lines 24-25)
5. As a process P_i executes, a "cache miss index" is also calculated from the number of cache misses and the number of cache references generated by P_i. This calculation is made asynchronously with respect to other concurrently executing processes. (Lines 26-27)
6. If IPC_i is lower than a threshold (< 1.7), or $fraction\_wait\_time_i$ is higher than a threshold (> 0.5), then the performance index for that process, $performance\_index_i$ is evaluated as true. The performance index of a process is evaluated as true if the process has decreased performance, caused by a decrease in its instructions per cycle rate, or due to longer periods of time spent by the

process in the ready or wait queues. We chose an IPC threshold of 1.7 based upon process performance data indicating that these processes had an IPC value greater than 1.7 during initial phases of execution. We chose a fraction wait time threshold of 0.5 because it is the average of 0, which signifies that a process is not spending any time waiting in the ready and I/O queues, and 1, which signifies that a process is spending all of its execution time waiting in the ready and I/O queues.

7. If $cache\_miss_i/cache\_reference_i$ is higher than a threshold (> 0.0014), then the cache miss index for that process, $cache\_miss\_index_i$, is evaluated as true. The cache miss index of a process is evaluated as true if the process generates a large number of cache misses with respect to the number of cache references generated. We chose a cache miss index threshold of 0.0014 because we wanted to use a threshold that would force our algorithm to alter its scheduling strategy if the ratio between the number of cache misses and cache references was greater than half of the average value for the cache miss index of all executing processes in our experimental group. Based upon experimental data, a value of 0.0014 represents this threshold.
8. At regular time intervals, for an executing process, $performance\_index_i$ is calculated. Each interval is equal to approximately one-fifth of the process's total execution time.

   a) If the process is CPU intensive (the average CPU utilization of all cores upon which it is executing is > 50 percent), its nice value is < 9, the performance index is true for that process, and if the cache miss index is false for that process, $load\_estimate_i$ is recalculated for all cpusets, and the process is context switched to the cpuset with the lowest value of $load\_estimate_i$. (Lines 29-35) A cpuset with a lower value of $load\_estimate_i$ will contain CPUs operating at higher frequencies and fewer processes in the ready queues of its cores. The goal of context switching a high priority, CPU intensive process that does not generate many cache misses, and that has decreased performance, to such a cpuset is to speed up the execution of that process. We chose a CPU utilization threshold of 50 percent to categorize a process as CPU intensive because we found, in our experiments, that the non-CPU intensive benchmarks used for our cases all had a CPU utilization less than 50 percent, and the test CPU intensive benchmarks all had a CPU utilization greater than 50 percent.

   b) To account for performance saturation caused by memory access, if a process is not CPU intensive, its nice value is > -1, and if the load average is $> num\_cpus$, or if a process is not CPU intensive, its nice value is ≤ -1, and the load average is $> 2 * num\_cpus$, then the process is context switched to the cpuset with the greatest value of $load\_estimate_i$. (Lines 36-50) If the system is heavily loaded and the process is not CPU intensive, the process is context switched to a cpuset with lower frequency cores. The higher the priority of the process, the more heavily loaded the system must be before that process is context switched. If the non-CPU intensive process's nice value is ≤ -11, the load average is $\leq num\_cpus$, if the performance index is true, and if the cache miss index for that process is false, or if the process's nice value is > -11, the load average is $< 0.15 * num\_cpus$, if the performance index is true, and if the cache miss index for that process is false, then this process is context switched to the cpuset with the lowest value of $load\_estimate_i$. (Lines 51-66) If the system is not heavily loaded, and a process does not generate a large number of cache misses and has decreased performance, then the process is context-switched to a cpuset with higher frequency cores. The lower the priority of the process, the lower the load average must be before the process is context switched.

9. For a non-executing process, if the fraction wait time of that process, $fraction\_wait\_time_i$, is ≥ 0.7 and its nice value is < -1, or if $fraction\_wait\_time_i$ of that process is > 0.9, then $load\_estimate_i$ is recalculated for all cpusets, and the process is assigned to the cpuset with the lowest value of $load\_estimate_i$. (Lines 7-18) If the process is waiting in the ready queues for the cores of a cpuset for too long, it is context switched to a cpuset with fewer processes in its ready queues. The higher the priority of the process, the less time it must spend waiting in the ready queues before it is context switched.
10. For all executing processes, a "CPU_intensity_priority_index" is calculated from the CPU intensity and nice value of that process.
11. At regular time intervals, the average number of threads in the ready queues of all cpusets is computed. If the average number of threads in the ready queues of any cpuset is < 1, then a process is assigned to that cpuset. If two or more cpusets have an average number of threads in their ready queues that is < 1, a process is first assigned to the cpuset with CPUs operating at the highest frequency. Process(s) are chosen to migrate to the new cpuset according to their CPU_intensity_priority_index, with those having a higher value of this index having higher priority. (Lines 69-75) If there is a cpuset containing a core with an empty ready queue, then a process is assigned to this core. Process(s) are chosen according to both their CPU intensity and process priority.
12. Thread parallelism is accounted for by the number and frequency of CPUs in cpusets (hardware), with the lack of parallelism available in cpusets with fewer CPUs compensated for by the higher frequencies of its CPUs.

13. The measure of the algorithm's effectiveness is:

    a) Average Instructions Per Second/Watts

$$= \left(\sum_{i=0}^{m} (Instruction\ Executed\ by\ P_i / Execution\ Time\ (second)\ for\ P_i)/\mathrm{m}/\mathrm{watts} \right)$$

    b) where m = number of processes in task T

    c) Average time required for all processes in task T to complete execution $*$ watts

$$= \left(\sum_{i=0}^{m} Execution\ Time\,(seconds)\,for\ P_i/\mathrm{m}^{*}\mathrm{watts} \right)$$

where $\mathrm{m} = \mathrm{number}$ of processes in Task T

(A potentially more efficient algorithm will have a higher value of (a) and a lower value of (b))

3 Feedback-Driven Priority-Based CPU Scheduling Algorithm 2

In this section we present another application-driven feedback mediated CPU scheduling algorithm, Algorithm *Context Switch Priority CPU Scheduler* (*CS-PCS*), that considers the number of context switches and CPU migrations generated at runtime instead of the number of cache misses. This algorithm also uses dynamic performance data and static or dynamic process priorities and is very similar to Algorithm *CM-PCS*. The only differences between both algorithms are on lines 26-29, 33-34, 56-57, and 65-66 of Algorithm *CS-PCS*. These differences are emphasized in bold font in both algorithms.

3.1 Variables and Constants Used in Algorithm CS-PCS

Algorithm *CS-PCS* uses the same data structures described in the previous section for Algorithm *CM-PCS* with the following exceptions. Instead of $cache\_miss_i$, $cache\_reference_i$, and $cache\_miss\_index_i$, Algorithm *CS-PCS* uses the following variables:

$context\_switch_i$ :: the number of context switches generated by process P_i;

$CPU\_migration_i$ :: the number of CPU migrations generated by process P_i;

$context\_switch\_CPU\_migration\_index_i$ :: a Boolean value calculated for an executing process P_i that uses $context\_switch_i$ and $CPU\_migration_i$;

$context\_switch\_CPU\_migration\_index_i$=true iff $(0.5 * \frac{context\_switch_i}{real\_time_i}) + (0.5 * \frac{CPU\_migration_i}{real\_time_i}) > 17.0$;

3.2 Algorithm CS-PCS

Algorithm 2: *Context Switch Priority CPU Scheduler* (*CS-PCS*)

```
1: Start with a set $S$ of processes $P_i$ in a given task $T$:
2: for all $P_i$ in $T$ (concurrently){

3:  //initially schedule a process
4:  calculate $load\_estimate_i$ for all cpusets;

5:  assign $P_i$ to cpuset with least value of
6:   $load\_estimate_i$;

7:  //if non-executing process is in wait queue
8:     too long, context switch process
9:  if($fraction\_wait\_time_i \geq 0.7$ and $nice_i < -1$){
10:       recalculate $load\_estimate_i$ for all
11:       cpusets and assign $P_i$ to cpuset with least
12:       value of $load\_estimate_i$;
13:    }
14:    else if($fraction\_wait\_time_i > 0.9$){
15:       recalculate $load\_estimate_i$ for
16:       all cpusets and assign $P_i$ to cpuset with
17:       least value of $load\_estimate_i$;
18:    }

19:  do{
20:    execute $P_i$;

21:    calculate $CPU\_intensity_i$ for $P_i$;

22:    calculate $IPC_i$ and $fraction\_wait\_time_i$
23:    for $P_i$;

24:    compute $performance\_index_i$ of $P_i$ using
25:     $IPC_i$ and $fraction\_wait\_time_i$;

26:    **compute**
27:     **$context\_switch\_CPU\_migration\_index_i$**
28:    **of $P_i$ using $context\_switch_i$ and**
29:     **$CPU\_migration_i$;**

30:    measure $load\_average$;

31:    if($CPU\_intensity_i > 50\%$ and $nice_i < 9$
32:       and $performance\_index_i$ == true and
33:        **$context\_switch\_CPU\_migration\_index_i$**
34:       == **false**){
35:          recalculate $load\_estimate_i$ for all
36:          cpusets and context switch $P_i$ to cpuset
37:          with least value of $load\_estimate_i$;
38:          }
39:  else if($CPU\_intensity_i < 50\%$ and $nice_i > -1$
40:       and $load\_average > num\_cpus$){
41:          recalculate $load\_estimate_i$ for all
42:          cpusets and context switch $P_i$ to
43:          cpuset with greatest value of
44:           $load\_estimate_i$;
45:       }
46:  else if($CPU\_intensity_i < 50\%$
47:       and $nice_i \leq -1$
48:       and $load\_average > 2 * num\_cpus$){
49:          recalculate $load\_estimate_i$ for all
```

TABLE 1
Dynamic Priorities of Benchmarks Evaluated

Benchmark	First Nice	First Priority	Last Nice	Last Priority
noncpuintensive$_{(1-4)}$	-16	1	-6	3
mcf$_{(1-4)}$	-10	2	0	5
gcc$_{(1-4)}$	-4	3	-14	1
bzip2$_{(1-4)}$	2	4	-8	2
sjeng$_{(1-4)}$	8	5	-2	4

```
50:          cpusets and context switch $P_i$
51:          to cpuset with greatest value of
52:          $load\_estimate_i$;
53:       }
53:   else if($CPU\_intensity_i$ < 50% and $nice_i \leq -11$
54:       and $load\_average \leq num\_cpus$
55:       and $performance\_index_i$ == true and
56:       context_switch_CPU_migration_index_i
57:       == false){
58:          recalculate $load\_estimate_i$ for all
59:          cpusets and context switch $P_i$ to cpuset
60:          with least value of $load\_estimate_i$;
61:    }
62:   else if($CPU\_intensity_i$ < 50% and $nice_i > -11$
63:       and $load\_average < 0.15 * num\_cpus$
64:       and $performance\_index_i$ == true and
65:       context_switch_CPU_migration_index_i
66:       == false){
67:          recalculate $load\_estimate_i$ for all
68:          cpusets and context switch $P_i$ to cpuset
69:          with least value of $load\_estimate_i$;
70:       }
71: //if a ready queue is empty, then assign a process
72:    to it
73:   if(a ready queue is empty and $P_i$ is process with
74:     highest value of $CPU\_intensity\_priority\_index_i$
75:     of all executing processes)
76:     {
77:        context switch $P_i$ to cpuset with the empty
78:        ready queue;
79:     }
80: }until $P_i$ terminates;
81:}
```

4 Evaluation and Results

For our research, we wished to investigate the performance gains or losses, in terms of both power and energy, and power savings achieved by Algorithms *CM-PCS* and *CS-PCS* when executing a typical workload. To this end, we wanted to include a combination of both CPU intensive and non-CPU intensive applications in our workload. Hence we tested our algorithm using five different benchmarks, four from the SPEC CPU2006 suite and one non-CPU intensive benchmark that we wrote. The SPEC CPU2006 benchmarks, representing a variety of architectural characteristics, were bzip2, gcc, mcf, and sjeng. Since all the benchmarks in the SPEC CPU2006 suite, including mcf, generated a CPU utilization above 85 percent, we also used a non-CPU intensive benchmark that we wrote in C. This benchmark repeatedly executed the "chmod" command and only generated a CPU utilization < 35 percent.

We utilized four of the SPEC CPU2006 benchmarks due to the following reasons. In preliminary studies, we measured the effects of using three concurrently executing benchmarks, two CPU intensive benchmarks, bzip2 and gcc, and one non-CPU intensive benchmark, noncpuintensive. We chose a similar combination for the five and eight concurrently executing benchmark test cases, and since most of the SPEC CPU2006 benchmarks are CPU intensive, we could only include a subset of them to ensure a combination of both types of benchmarks in our test cases. In this paper, we wanted to assess the results for a task which represents a more realistic load for a typical server, namely a task containing more than eight concurrently executing processes. However, we wanted to use the same benchmarks that we used before, that is, those used for tasks containing three, five, and eight concurrently executing processes, in order to eliminate as many experimental variables as possible when comparing a heavily loaded system to a lightly loaded system, which is anticipated in an extension of this work.

In this research, we ran a task composed of three separate instances of bzip2, three separate instances of gcc, three separate instances of mcf, three separate instances of sjeng, and twelve separate instances of noncpuintensive, for a total of twenty-four benchmarks. To derive a more accurate estimate of the performance indicators, we took an average of the performance data for five executions of each benchmark. Thus the benchmarks within a task were restarted once they finished until each benchmark completed execution five times. The benchmarks in one task were run concurrently, while those in separate tasks (for either the test or control group) were not run concurrently.

We used two experimental groups, a test group in which a task was scheduled using either Algorithm *CM-PCS* or Algorithm *CS-PCS*, and a control group in which the same task was scheduled using the default Linux scheduler and "on demand" CPU frequency scaling governor. Hence each task was executed twice. We chose the "on demand" governor for our control group because it is the default CPU frequency scaling governor in the Fedora Core 14 package, but more importantly, because we wanted to

TABLE 2
Average Number of Cache Misses, Cache References, and Average Cache Miss/Reference Ratio for Algorithm 1 (*CM-PCS*), Algorithm 2 (*CS-PCS*), and Control

	Alg. 1	Alg. 2	Control
Avg. # of Cache Misses (in millions)	139.00	143.83	120.40
Avg. # of Cache References (in billions)	49.45	49.58	48.86
Avg. Cache Miss/Reference Ratio	.0028	.0029	.0025

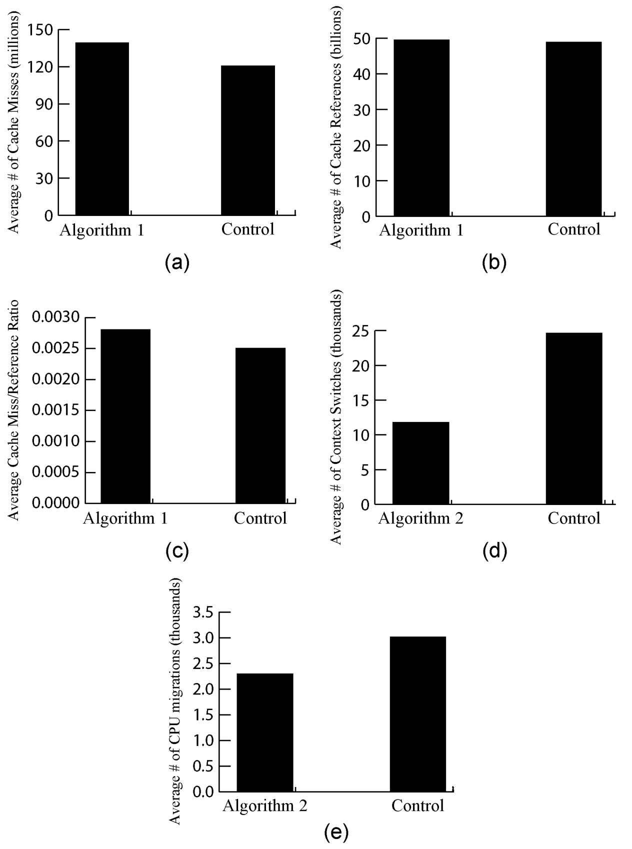


Fig. 1. Average number of cache misses, cache references, and cache miss/reference ratio for algorithm 1 (*CM-PCS*) over Control and context switches and CPU migrations for algorithm 2 (*CS-PCS*) over control. (a) Average number of cache misses for algorithm 1 and control; (b) average number of cache references for algorithm 1 and control; (c) average cache miss/reference ratio for algorithm 1 and control; (d) average number of context switches for algorithm 2 and control; (e) average number of cpu migrations for algorithm 2 and control.

compare the effects of using our algorithms versus a governor that would employ a different approach to advantageously using CPU frequency scaling.

In our test case, we assigned dynamic priorities to our processes, and in order to test a more realistic load for a server, we used twenty-four concurrently executing benchmarks. When assigning dynamic priorities, the first priority (Priority 1) was assigned to a process for its first three execution cycles, and this process's second priority (Priority 2) was assigned during its last two execution cycles. Also, as shown in Table 1, these priorities were unique, with the exception of those assigned to multiple instances of identical benchmarks. Without loss of generality, a lower nice value implies a higher priority and a higher nice value implies a lower priority, as shown in Table 1. Our control group consisted of the same processes, having the same priorities as the test group, executing with the default Linux scheduler and CPU frequency scaling governor.

Table 2 and Fig. 1c show that the average cache miss/reference ratio for Algorithm *CM-PCS* was very close to that obtained by the control group and lower than that of Algorithm *CS-PCS*. This indicates that despite the extra context switches generated by Algorithm 1 in its attempt to move more CPU intensive processes to higher frequency cores, this increase in context switching is controlled by the algorithm's use of the cache miss index, resulting in a concomitant reduction of the cache miss to cache reference ratio.

The use of the cache miss/cache reference ratio by Algorithm 1 to reduce cache miss and context switching overhead does lead to increased performance. As shown in Table 3, performance, in terms of average instructions per second, and performance per watt were both higher for the test group than the control group by Algorithm *CM-PCS*. Also, the metric execution time · watt (or energy consumption) was lower for the test group than the control group, as indicated by Table 3.

Also, Table 4 shows that there is a significant increase in the percent power savings and percent improvement of performance, performance per watt, and execution time · watt (energy) for Algorithm *CM-PCS* when compared to control. Also, there is a very significant total power savings of 39 watts.

Table 6 and Figs. 1d and 1e show that the average number of context switches and CPU migrations for Algorithm *CS-PCS* was lower than that of Algorithm *CM-PCS* and much lower than that of control. Also, as shown in Table 3, performance, in terms of average instructions per second,

TABLE 3
Performance, Performance per Watt, and Execution Time · Watt (Energy) for Algorithms 1 (*CM-PCS*) and 2 (*CS-PCS*) and Control

	Alg. 1 test	Alg. 2 test	control
Performance Avg Ins/Sec (in billions)	0.69	0.79	0.67
Performance per Watt Avg (Ins/Sec)/Watt (in millions)	4.86	5.46	3.66
Execution Time · Watt (Energy) Avg Time (Sec) · Watt (in thousands)	32.45	28.86	38.26

TABLE 4
Total Power Savings, Percent Power Savings, and Percent Improvement of Performance, Performance per Watt, and Execution Time · Watt (Energy) by Algorithms 1 (*CM-PCS*) and 2 (*CS-PCS*)

	Alg. 1	Alg. 2
Total Power Savings (Watts)	39	38
Power Savings (%)	21.43	20.88
Performance Improvement (%)	4.21	15.34
Performance per Watt Improvement (%)	24.74	33.02
Execution Time · Watt (Energy) Improvement (%)	15.18	24.56

TABLE 5
Percent Increase of Average Number of Cache Misses, Cache References, and Average Cache Miss/Reference Ratio over Control by Algorithm 1 (*CM-PCS*)

	Percent Increase Over Control by Algorithm 1
Avg. Number of Cache Misses (%)	13.38
Avg. Number of Cache References (%)	1.18
Avg. Cache Miss to Cache Reference Ratio (%)	10.71

TABLE 6
Average Number of Context Switches and CPU Migrations for Algorithm 1 ($\mathcal{CM}$-*PCS*), Algorithm 2 ($\mathcal{CS}$-*PCS*), and Control

	Alg. 1	Alg. 2	Control
Avg. # of Context Switches (in thousands)	15.85	11.76	24.59
Avg. # of CPU Migrations (in thousands)	2.95	2.29	3.01

and performance per watt, by Algorithm $\mathcal{CS}$-*PCS*, were both higher for the test group than the control group. Also, Table 3 indicates that the metric execution time · watt (or energy consumption) was lower for the test group than the control group. These differences were even more significant than those seen by Algorithm $\mathcal{CM}$-*PCS*.

In addition, Table 4 shows that there is a very significant increase in the percent power savings and percent improvement of performance, performance per watt, and execution time · watt (or energy) for Algorithm $\mathcal{CS}$-*PCS* when compared to control. In fact, the algorithm created a power savings which was as high as 38 watts while simultaneously generating a performance gain of 15.34 percent, for tasks composed of twenty-four concurrently executing benchmarks. This led to a very significant improvement of the performance per watt metric (33.02 percent).

As shown by Table 7, there is a greater than 50 percent reduction in the average number of context switches by Algorithm 2 over control and a very significant reduction in the average number of CPU migrations over control by Algorithm $\mathcal{CS}$-*PCS*. Since there is also a very significant improvement in performance seen by Algorithm $\mathcal{CS}$-*PCS*, this implies that the reduction of context switching overhead generated by this algorithm is high enough to increase performance but is not so high as to stifle the context switching necessary to migrate processes that can benefit from executing upon higher frequency cores.

5 Conclusion

The main motivation of our research was to design methods to allocate CPU resources in a multicore processor system with the goal of lowering the global power budget and creating a minimal performance loss (or a performance gain). To accomplish this goal, we used "application-driven" feedback, in which an executing application gave feedback (regarding its performance) to the operating system at runtime, and the operating system, in turn, dynamically scheduled the system's CPU hardware resources based upon this feedback. We presented two CPU scheduling algorithms that create and utilize cpusets containing varying numbers of cores/processors of varying frequencies, and, in doing so, lower the global power budget in a multicore or multiprocessor system, while simultaneously creating a performance gain.

TABLE 7
Percent Reduction of Average Number of Context Switches and CPU Migrations over Control by Algorithm 2 ($\mathcal{CS}$-*PCS*)

	Percent Reduction Over Control by Algorithm 2
Average Number of Context Switches (%)	52.19
Average Number of CPU Migrations (%)	24.15

Our algorithms not only match a process to cores better suited to execute this process based upon its performance characteristics, but also consider the performance characteristics of the task as a whole, process priority, and system load. They also use a unique method of initial task assignment, to avoid large initial load imbalances. By utilizing "cache miss" and "context switch-CPU migration" indices, our algorithms are able to exploit the increased performance associated with context switching more CPU intensive processes to higher frequency cores, without suffering from the performance loss associated with cache coherence and context switching overhead. Consequently, we have demonstrated that the implementation of our algorithms not only results in significant power savings, but for a multicore system containing twenty-four concurrently executing benchmarks, also simultaneously results in a significant improvement in performance, performance per watt, and execution time · watt (or energy consumption). We have also demonstrated a total power savings of up to 39 watts, or 21.43 percent, and a performance gain of 15.34 percent, when using our algorithms compared to using the default Linux scheduler and "on demand" CPU frequency scaling governor.

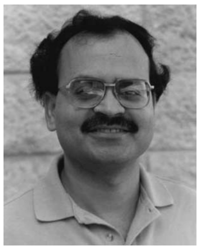
In our research, we have tested tasks composed of multithreaded processes and those containing both CPU intensive and non-CPU intensive processes. We have shown, using real hardware and not just simulation, that our CPU scheduling algorithms are superior to the default Linux scheduler and CPU frequency scaling governor in terms of both performance and power savings for a task composed of twenty-four concurrently executing benchmarks.

The significance of this research is that in a data center, like the server systems used for cloud computing, even a modest power savings, and certainly a power savings of 39 watts, for each server can result in a huge power savings for the data center as a whole. Also, decreased power dissipation can result in decreased heat dissipation, further reducing power usage and increasing CPU and hardware component life. Finally, a decrease in the power consumption of CPUs is important for other applications such as notebook computers, where prolonging battery life is crucial and cooling hardware components may be difficult.

There are many future extensions of this work. An obvious extension is to use a manycore chip, one with tens, hundreds, or even thousands of cores, in our research. We may also consider measuring the effect of our algorithms upon a lightly loaded system (one with three, five, or eight concurrently executing benchmarks) and comparing those results to the ones presented in this paper, in which we surveyed a heavily loaded system. Finally, we may consider different core partitioning strategies, with the goal of spreading heat dissipation and dispersing thermal hot spots or increasing performance.

REFERENCES

[1] Wikimedia Foundation, Inc., 2011. [Online]. Available: http://en.wikipedia.org/wiki/Pentium4
[2] Wikimedia Foundation, Inc., 2011. [Online]. Available: http://en.wikipedia.org/wiki/Multi-coreprocessor
[3] D. Shelepov and A. Fedorova, ''Scheduling on Heterogeneous Multicore Processors Using Architectural Signatures,'' in *Proc. Workshop Interact. Oper. Syst. Comput. Architect.*, 2008, pp. 1-9.
[4] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen, ''Single-ISA Heterogeneous Multicore Architectures for Multithreaded Workload Performance,'' in *Proc. 31st Annu. Int. Symp. Comput. Architect.*, 2004, pp. 64-75.
[5] Intel Corporation, Santa Clara, CA, USA, 2010. [Online]. Available: http://software.intel.com/en-us/articles/detecting-memory-bandwidth-saturation-in-threaded-applications/
[6] S. Ghiasi, T. Keller, and F. Rawson, ''Scheduling for Heterogeneous Processors in Server Systems,'' in *Proc. Comput. Frontiers. ACM Press*, 2005, pp. 199-210.
[7] G.M. Almeida, R. Busseuil, E.A. Carara, N. Hebert, S. Varyani, G. Sassatelli, P. Benoit, L. Torres, and F.G. Moraes, ''Predictive Dynamic Frequency Scaling for Multi-Processor Systems-on-Chip,'' in *Proc. IEEE ISCAS*, May 2011, pp. 1500-1503.
[8] C. Isci, A. Buyuktosunoglu, C.Y. Cher, P. Boise, and M. Martonosi, ''An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget,'' in *Proc. 39th Annu. IEEE/ACM Int'l Symp. Microarchitect.*, 2006, pp. 347-358.
[9] A. Vajda, ''Space-Shared and Frequency-Scaling Based Task Scheduler for Many-Core OS,'' in *Proc. Workshop HotPower Aware Comput. Syst. SOSP*, 2009, pp. 1-5.
[10] A. Fedorova, D. Vengerov, and D. Doucette, ''Operating System Scheduling on Heterogeneous Core Systems,'' in *Proc. Oper. Syst. Support Heterogeneous Multicore Architect.*, 2007, pp. 1-9.



Ajoy K. Datta is a Professor of Computer Science at the University of Nevada-Las Vegas, Las Vegas. His research interests are in the areas of distributed computing and self-stabilization.

Rajesh Patel received the PhD degree in computer science at the University of Nevada-Las Vegas, Las Vegas, in 2012. He works in the software development field as a. NET Developer. His research interests are in the fields of sensor networks, muticore computing, and CPU scheduling.