

# Operating System Scheduling On Heterogeneous Core Systems

Alexandra Fedorova  
Simon Fraser University  
fedorova@cs.sfu.ca

David Vengerov  
Sun Microsystems  
David.Vengerov@sun.com

Daniel Doucette  
Simon Fraser University  
ddoucett@cs.sfu.ca

## Abstract

*We make a case that a thread scheduler for heterogeneous multicore systems should target three objectives: optimal performance, core assignment balance and response time fairness. Performance optimization via optimal thread-to-core assignment has been explored in the past; in this paper we demonstrate the need for balanced core assignment. We show that unbalanced core assignment results in completion time jitter and inconsistent priority enforcement; we then present a simple fix to the Linux scheduler that eliminates these problems. The second part of the paper addresses the problem of building the HMC scheduler that balances all three objectives. This is a difficult optimization problem. We introduce a definition of this scheduling problem in terms of these three objectives and present a blueprint for a self-tuning algorithm based on reinforcement learning that maximizes a performance function that is an arbitrary weighted sum of these three objectives. Implementing and evaluating this algorithm is the subject of our future work.*

## 1. Introduction

Operating system (OS) schedulers for conventional processors are tuned for response time fairness: they distribute CPU time among the jobs in some fair fashion and limit the delay associated with waiting for CPU [3,6]. The advent of heterogeneous multicore (HMC) processors motivates new performance considerations for scheduling algorithms. On a HMC system, performance can be improved by scheduling a job to run on the core that is best suited for that job – i.e., the job’s *preferred* core [5]. This performance objective may conflict with the response time fairness objective of conventional schedulers. Consider the following example: Suppose that at the time when the scheduler dispatches thread  $A$  to run on a processor,  $A$ ’s preferred core is occupied by another running thread. The scheduler faces a dilemma: should it

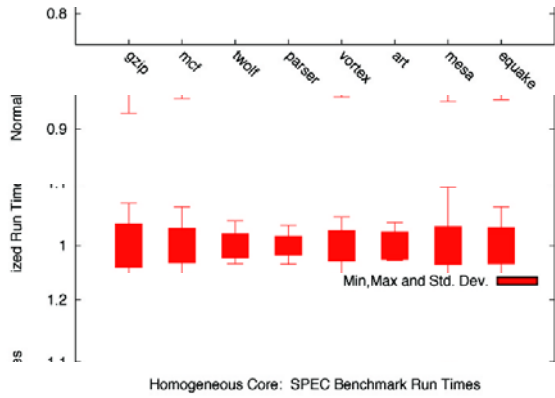
schedule  $A$  to run on its non-preferred core (that is available right away) or have it wait until its preferred core becomes available? Making  $A$  wait will potentially increase  $A$ ’s response time and reduce its allocated fraction of CPU cycles (compared to a conventional, homogeneous system). On the other hand, scheduling  $A$  on a non-preferred core will produce sub-optimal instruction throughput for  $A$ . Making the right decision requires carefully weighing performance/fairness tradeoffs and making the choice that meets system’s goals with respect to both objectives.

The scheduler must also consider *core assignment balance*. Modern thread schedulers are pre-emptive, and so a thread’s execution time will be usually spread among the system’s cores, but not necessarily in a balanced fashion. That is, a thread may execute a larger fraction of time on one core than on another. On a HMC system, such unbalanced core assignment will result in jittery performance (as was shown in the past [2] and validated again in this paper), and inconsistent priority enforcement (as demonstrated for the first time in this paper).

While previous work demonstrated that system-wide instruction throughput can be improved by a scheduler that considers optimal core thread-to-core assignment [5], balancing this optimization goal with response time fairness and core assignment balance has not been addressed. We begin addressing this problem in this study.

The contributions of this paper are as follows:

- (1) We show experiments demonstrating that unbalanced core assignment leads to completion time jitter and inconsistent priority enforcement;
- (2) we describe and evaluate a fix to the Linux scheduler that enforces balanced core assignment;



**Figure 1.** SPEC benchmark runtimes in the homogeneous setup.

- (3) we present a formal definition of a HMC scheduling problem in terms of balancing three objectives: optimal performance, response time fairness and core assignment balance;
- (4) we present a potential solution to the problem: a self-tuning scheduling algorithm based on reinforcement learning. Implementing and evaluating that algorithm is in our plans for future work.

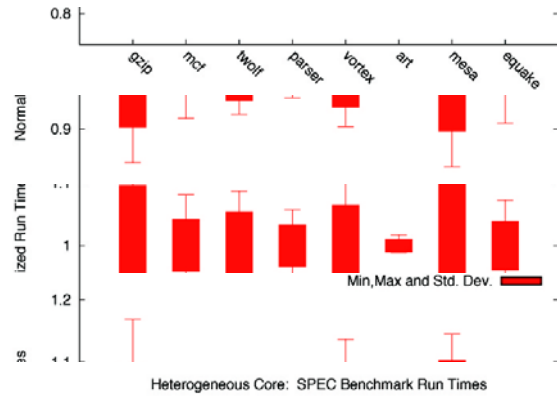
The rest of the paper is organized as follows: We present the four contributions outlined above in sections 2-5 respectively. In Section 6 we present related work, and in Section 7 we summarize.

## 2. Effects of Unbalanced Core Assignment

In this section we present results of experiments demonstrating that unbalanced core assignment on a HMC system results in completion time jitter (Section 2.2) and inconsistent priority enforcement (Section 2.3). The jitter phenomenon has been demonstrated before for multithreaded workloads [2], and here we re-validate these results for multi-program workloads. Results on inconsistent priority enforcement are new.

### 2.1. Experimental setup

We run our experiments on a system equipped with two 2.8 GHz Intel® Xeon™ hyper-threaded processors (Northwood series), running the ubuntu distribution of Linux with 2.6.15 version of the kernel. We disabled hyper-threading. To create a heterogeneous system, we scaled down the clock speed on one of the CPUs using the mechanism available in Xeon processors (intended for thermal



**Figure 2.** SPEC benchmark runtimes in the heterogeneous setup.

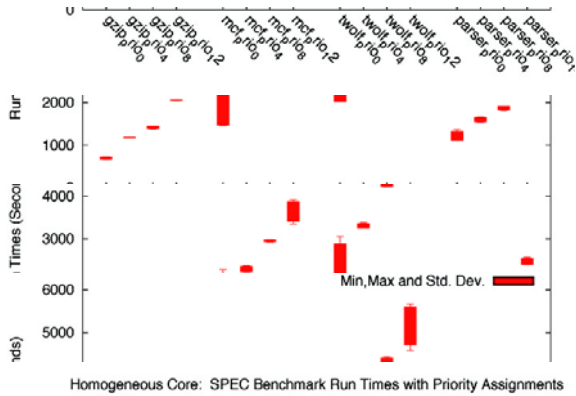
management). The same method was used in an earlier study to create a heterogeneous system [2]. In our heterogeneous setup, one processor is running at 2.8 GHz, and the other one at 1.05 GHz. In the homogeneous setup, both processors are running at 2.8 GHz.

### 2.2. Completion time jitter

We selected eight benchmarks from the SPEC CPU2000 suite [1] representing a variety of architectural characteristics: *gzip*, *mcf*, *twolf*, *parser*, *vortex*, *art*, *mesa*, and *equake*. We run these benchmarks simultaneously; we restart each benchmark once it is finished until each benchmark completes at least five times. For each benchmark, we measure the minimum and maximum running time (i.e., completion time), and the standard deviation from the average running time.

Figure 1 shows these measurements, normalized against the mean, for the homogeneous setup, and Figure 2 for the heterogeneous setup. The mean running time corresponds to “1” on the Y-axis. The box boundaries indicate the standard deviation. The whiskers correspond to the minimum and maximum running times.

The results indicate that in the heterogeneous setup the running times are less predictable, more jittery, than in the homogeneous setup. In the homogeneous setup, the minimum and maximum running times are much closer to the mean than in the heterogeneous setup. Standard deviations in the homogeneous setup are also smaller. (The exception is *art*, which experiences less jittery performance in the heterogeneous setup than in the homogeneous setup;



**Figure 3.** SPEC benchmark runtimes for varying priorities in the homogeneous setup.

we do not yet understand this phenomenon and are investigating it.)

Applications suffer from jittery completion times on heterogeneous multicore systems, because the application’s instruction rate varies from one core to another, and when the core assignment is unbalanced the overall running time depends on the particular core assignment during the run.

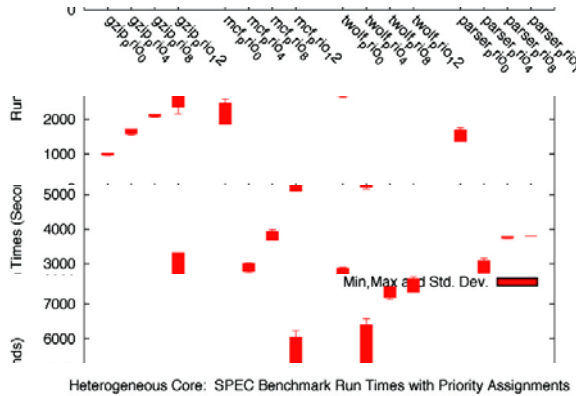
The earlier study that demonstrated performance jitter [2] explained its presence by the fact that the OS was inconsistent as to which of the cores was left idle whenever the system load decreased such that not all cores were used to run application threads. This explanation does not apply to our experimental environment, because the cores were never left idle during the experiments. In our case, jitter was caused by unbalanced core assignment, and we confirm this (in Section 3) by showing that a simple scheduler fix that enforces balanced core assignment eliminates jitter.

**2.3. Inconsistent priority enforcement**

In this section we demonstrate a new result with respect to performance on HMC systems. We show that unbalanced core assignment results in inconsistent priority enforcement.

We present results for the first four of our eight benchmarks: *gzip*, *mcf*, *twolf*, and *parser*. We could not gather data for all eight benchmarks due to lack of time.

The experiments were run as follows: we ran each benchmark with four different *nice* settings: 0, 4, 8,



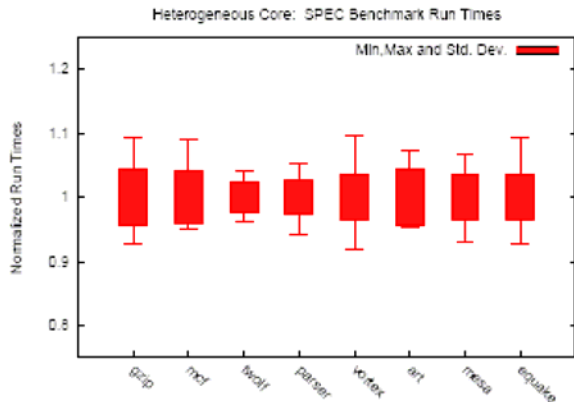
**Figure 4.** SPEC benchmark runtimes for varying priorities in the heterogeneous setup.

and 12. This resulted in four different priority levels for the benchmark, level 0 corresponded to the highest priority. Each benchmark was run five times with each priority level. For each priority level, we measured the mean, maximum and minimum running time and the standard deviation from the mean. Along with the measured benchmark, we ran the remaining three benchmarks at the default priority level.

Figure 3 shows the results for the homogeneous setup, Figure 4 for the heterogeneous setup. Priorities are enforced less consistently in the heterogeneous setup. For all benchmarks except *mcf*, the distinction between running times for different priority levels of the same benchmark is not as clear in the heterogeneous setup as in the homogeneous setup.

Examining *gzip* in the homogeneous setup, we can see that its running time steadily increases as its nice level increases. In the heterogeneous setup, this is not always the case. For example, the minimum running time for nice level 12 is the same as the average running time for nice level 8. The scheduler does not consistently enforce the user-assigned priority.

Looking at *twolf* and *parser*, in the homogeneous setup the difference in running times for nice levels 8 and 12 is obvious; in the heterogeneous setup it is not possible to distinguish between the running times at these two different priority levels.



**Figure 5.** SPEC benchmark runtimes for varying priorities on the heterogeneous setup with the new scheduler.

Such inconsistent priority enforcement is the direct consequence of jittery running times caused by unbalanced core assignment.

### 3. Enforcing Core Assignment Balance

In this section we describe our enhancement to the Linux scheduler that enforces balanced core assignment. The new scheduler ensures that each thread’s execution time is divided evenly across all system’s cores. This reduces variation in the job’s instruction throughput from run to run, and as a result reduces running time jitter.

The new scheduling algorithm works as follows. On each scheduler tick, the CPU inspects the run queue of the CPU one number higher than the current CPU to see if there are any jobs that have completed their timeslice on that CPU. (The highest numbered CPU examines the lowest numbered CPU.) If such jobs are found, they are moved from the inspected CPU to the inspecting CPU. That is, CPU 0 inspects CPU 1, CPU 1 inspects CPU 2, ... , CPU N inspects CPU 0. The CPU detects the job that should be moved by examining the 'hcs\_last\_cpu' variable of each job: that variable is set to be equal to the CPU where the thread ran during its last timeslice.

To dynamically enable and disable this core balancing feature we added a new global variable 'hcs\_balance\_enabled', and an entry to procs that is used to control that variable.

Figure 5 shows the normalized running times in the heterogeneous setup with the new scheduler. This is the same experiment that we described in Section

2.2. Figure 5 should be compared with Figure 2, which shows the same data collected with the old scheduler. We can see that the new scheduler significantly reduces completion time jitter. Comparing Figure 5 with Figure 1, we can see that running time jitter in the heterogeneous setup with the new scheduler is comparable to the homogeneous setup. (This applies to all benchmarks except *art* – we are still investigating the cause for the unexpected behavior in *art*.)

Balakrishnan et al. presented another fix to the Linux scheduler aimed at reducing performance jitter [2]. Their fix ensured that the scheduler always left the slower core idle, whenever there were not enough runnable threads to occupy all the cores. That reduced jitter only for some of their benchmarks; it did not apply to those benchmarks that never left any cores idle. Our fix works regardless of the presence of idle cores.

Our simple scheduler could be improved by considering cache affinity. In the current design, the scheduler moves the task to a new core after each timeslice. This could decrease performance if the task has some relevant data left in the old core’s cache: scheduling the task on the old core, rather than the new one, would have allowed the task to re-use that data.

Although our simple modification to the scheduler fixed runtime jitter, it did not target performance improvements that could be realized by considering optimal thread-to-core assignment. This consideration is relevant in HMC systems, where cores differ not just in speed, but in their features, and the core’s features determine the instruction rate produced by threads on that core [5]. On these systems, assigning each thread to the core best suited for it results in improved performance.

In the next section we formulate the problem of scheduling on HMC systems in terms of balancing these objectives, and in Section 5, we outline a potential solution.

### 4. The Scheduling Problem

A scheduler for HMC systems should balance between the following objectives:

- (1) Optimal performance
- (2) Core assignment balance
- (3) Response time fairness.

In this section we define the metrics to represent these objectives. In the Section 5 we present a self-tuning scheduling algorithm that maximizes performance function that is an arbitrary weighted sum of these three objectives.

#### 4.1. Optimal performance

We measure performance in terms of *system-wide rate of instructions*, that is, aggregate instruction per cycle rate (IPC) achieved by all running threads. Alternatively, performance could be expressed as the weighted speed-up over the worst-suited core: the sum of all jobs' performance improvements over the worst-suited core for each job. We believe that our self-tuning algorithm will be flexible enough to adapt to any suitable definition of performance. For the purposes of this paper, we assume that our performance goal is to maximize the system-wide rate of instructions.

#### 4.2. Core assignment balance

Core assignment balance ensures that a job's execution time is spread across all cores evenly. For a given job, we can measure the fraction of time that the job spent running on each core. Then we can measure the standard deviation from the average fraction – this is the metric of core assignment balance. We refer to this metric for job  $J$  as  $CAB_J$ .

DEFINITION 1: *Core assignment for a job  $J$  is considered balanced if:  $CAB_J \leq \varepsilon$ .*

A range of acceptable values for  $\varepsilon$  will be determined experimentally.

DEFINITION 2: *System-wide core assignment balance (CAB) is the percent of jobs satisfying the balance constraint in Definition 1.*

In the scheduler, the maximization of system-wide core assignment balance will be traded off against maximization of system-wide instruction rate and response time fairness.

A scheduler configured with a tight CAB constraint  $\varepsilon$  would tend to rotate jobs among the system's cores (like our simple scheduler presented in Section 3). A potential concern is that this would violate cache affinity. To prevent this problem, we consider cache affinity in our self-tuning algorithm, as will be explained in Section 5.

#### 4.3. Response time fairness

Our metric for response time fairness is based on a metric presented by Wierman and Harchol-Balter [11]. The Wierman-Harchol-Balter metric is based on a job's *slowdown* (defined below); the system is fair if each job's slowdown stays within bounds proportional to system load.

Slowdown for a job of size  $x$ ,  $S(x)$ , is defined as the job's response time  $T(x)$  divided by its size:

$$S(x) = \frac{T(x)}{x} \quad . \quad (1)$$

$T(x)$  is the sum of the time that the job waits for CPU and the time that the job runs on CPU. (The latter is proportional to the job size  $x$ .)

DEFINITION 3: *A job is treated fairly if its expected slowdown is within some constraint  $F$ :  $E[S(x)] \leq F$ .*

$F$  should be some function of system load. We will determine a good setting for  $F$  experimentally. Since our goal is to design a scheduler that accomplishes similar response time fairness as conventional thread schedulers, e.g., time-sharing schedulers in the Solaris<sup>TM</sup> and Linux operating systems, we will measure bounds on slowdown achieved by these schedulers on different loads, and derive  $F$  based on that data.

DEFINITION 4: *System-wide response-time fairness (RTF) is the percent of jobs satisfying the fairness constraint in Definition 3.*

In the rest of this section we discuss how we model jobs in our system to enable online measurement of job slowdown (this is needed by our algorithm to ensure that slowdown is kept within the bounds). Then we discuss the suitability of our response time fairness metric for our target environment.

#### 4.3.1. Job model

To compute response time fairness, we need to compute each job's slowdown. Computing the slowdown requires knowing the job's size, i.e., its processing requirements. In the environments that we target, that is, modern systems running modern applications, jobs' processing requirements are not known ahead of time. For example, server applications are typically started to run for an indefinite period of time until explicitly terminated by the user or until they crash. Therefore, we introduce a new technique to model jobs in our system; this technique will permit online measurement of the job's slowdown despite knowing its size ahead of time.

In our target environment, each new job  $J_k$  is mapped to a thread  $T_k$ .  $T_k$  is assigned a CPU timeslice and enqueued into a run queue, where it waits for its turn to run. Once its turn arrives,  $T_k$  runs until its timeslice expires or until it blocks, and then relinquishes the processor. After that,  $T_k$  is assigned a new timeslice (whose length could be different from the length of the old timeslice) and put back on the run queue.

Accordingly, we represent each job  $J_k$  as a sequence of sub-jobs  $\{j_0, j_1, \dots, j_N\}$ , where  $j_i$  is the portion of  $J_k$  running during the  $i$ th timeslice of  $T_k$ . Therefore, we can think of the length of the  $i$ th timeslice as being the size of  $j_i$ .

According to this job model, we update the definition of system-wide fairness:

**DEFINITION 4A:** *System-wide response-time fairness (RTF) is the percent of sub-jobs satisfying the fairness constraint in Definition 3.*

#### 4.3.2. The goodness of the fairness metric

Satisfying the fairness constraint given in Definition 3 implies that shorter jobs will wait for CPU less time than longer jobs (this follows from the definition of slowdown). In other words, threads with shorter timeslices will wait for CPU less than threads with longer timeslices. Time-sharing schedulers in Solaris and Linux use similar rules for thread scheduling, so we think that our fairness constraint will help us achieve similar response time fairness as those schedulers.

The Solaris operating system scheduler [6] assigns a higher dynamic priority to threads with shorter timeslices, so they wait for CPU less than threads with longer timeslices.

Linux scheduler [3] assigns a higher dynamic priority to threads that used a smaller portion of its assigned time quantum in a given scheduling epoch. (An epoch involves running all threads until they complete its assigned quantum.) Therefore, threads that usually give up the CPU quickly, i.e., threads whose timeslice is usually short, wait for CPU less than threads whose timeslice is usually long.

## 5. Self-Tuning Scheduling Algorithm

We propose a framework and a self-tuning algorithm for thread scheduling on HMC systems, which allows the system to *learn* the thread scheduling policy that maximizes the system's performance (system-wide instruction rate) and maintains fairness. Such an algorithm is required for finding the optimal thread allocation policy because the system's state changes stochastically over time (threads get blocked occasionally and go to sleep, the instruction rate of a thread changes over time as the work the thread is doing changes, etc.), and so the correlation between states, actions and system's performance needs to be learned through interaction with the system.

In our framework, each system core  $i$  learns a *benefit function*  $B_i()$  that approximates the expected future instruction rate on that core. The function  $B$  is used to make core assignment decisions.

The benefit function is based on some variables and tunable parameters (explained below); tunable parameters are adjusted after every thread migration decision based on the *reinforcement signal* (feedback), which is the value of the instruction rate on that core evaluated over the time between two consecutive thread allocation decisions. The objective of parameter tuning is to make  $B$  a better approximation to the expected future value of the core instruction rate.

In the remainder of this section we explain how  $B$  is computed (Section 5.1), how  $B$  is used to make core assignment decisions (Section 5.2), and how the

system updates tunable parameters of  $B$  (Section 5.3).

### 5.1. Computing $B$

The function  $B_i$  uses several variables as inputs, and the first one is the average value of *normalized core preference (NCP)* among threads on core  $i$ . The *NCP* of a thread  $j$  on core  $i$  is computed as follows:

$$NCP_{j,i} = \frac{IPC_{j,i}}{\max(IPC_{j,k})} \quad (2)$$

where  $IPC_{j,i}$  is the instructions per cycle (IPC) measure for thread  $j$  on core  $i$ , which is updated using the hardware instruction counter for all threads on core  $i$  whenever the time comes to perform thread migrations on that core;  $\max(IPC_{j,k})$  is the maximum IPC recently observed for thread  $j$  across all cores.

Another input variable for the function  $B_i$  is the average *cache affinity* of threads on core  $i$ . If a thread has executed on the core  $i$  within the last *rechoose\_interval* (a tunable parameter that is set by default to 0.03 seconds in Solaris), cache affinity is 1; otherwise, cache affinity is zero. The parameter-tuning algorithm should learn that it is more preferable to have threads with cache affinity 1 on a core, as it will mean that threads are using their cache investment and are executing more efficiently.

Note that if  $K$  threads have executed on a core within the last *rechoose\_interval* seconds, it still does not mean that all of them have their full cache investment intact, since the last few of these  $K$  threads could have displaced the cache investment of the previous threads if they had a high cache miss rate. Therefore, a third variable is needed in order to estimate the average cache investment of threads on a core: the average cache miss rate of threads on that core, which can be regularly updated for each thread using the hardware cache miss rate counter.

Let  $s_t$  be the value of the *state* vector composed of the three variables described above at time  $t$ . The benefit function  $B_i$  will have the following form:

$$B(s_t, p) = \sum_{n=1}^N p_n \varphi_n(s_t), \quad (3)$$

where  $\varphi_n(s)$  are pre-specified *basis functions* defined on the space of possible values of  $s$ , and  $p_n$ ,  $n=1, \dots, N$ , are the tunable parameters that are adjusted in the course of learning.

### 5.2. Using $B$ to make core assignment decisions

At regular time intervals the scheduler decides whether to migrate some number of threads among the cores. For each core  $i$ , it computes the value of  $B_i$  for the case when no thread migrations are performed:  $B_{i,0}$ . Then it selects some threads that can potentially be migrated to other cores (e.g., a thread that is currently assigned to its non-preferred core or a thread that has least cache affinity on that core). In the extreme case, all threads can be selected for potential migration to other cores. Then, for each considered thread  $k$  the scheduler computes the value of  $B$  that would arise if thread  $k$  were migrated to another core:  $B_{i,k}$  (by first computing the alternate state vector  $s'_i$  that would arise if the thread  $k$  were migrated to another core). Then, the scheduler chooses another core  $j$  (by selecting  $j$  to be thread  $k$ 's preferred core or by considering sequentially all cores) and computes the value  $B_{j,0}$  and also the value  $B_{j,k+}$ , which corresponds to the case when thread  $k$  is migrated from core  $i$  to core  $j$ .

Finally, thread  $k$  is migrated from core  $i$  to core  $j$  if:

$$B_{i,k-} + B_{j,k+} > B_{i,0} + B_{j,0} + a \cdot D_{CAB} + b \cdot D_{RTF}, \quad (4)$$

where  $D_{CAB}$  is the expected change in the system-wide CAB measure (per Definition 2) and  $D_{RTF}$  is the expected change in the system-wide RTF measure (per Definition 4A) that will occur as a result of this migration, and  $a$ ,  $b$  are coefficients that show the relative importance of these considerations and that are set *a priori* by a system administrator.

Computing  $D_{CAB}$  requires estimating the system-wide CAB that will occur as a result of the migration. This is the CAB that would result if all threads executed on the assigned core for the duration of its assigned timeslice. Similarly, computing  $D_{RTF}$  requires estimating the system-wide RTF that will occur as a result of the migration. This is the RTF that would result if all threads executed on the assigned core for the duration of its next assigned timeslice.

The migration criterion 4 implies that a particular thread migration is performed if the benefit of migrating the thread to another core is greater than not doing anything, provided that no large negative impact on the system-wide measures CAB and RTF will arise as a result of this migration.

### 5.3. Updating tunable parameters

In order for the migration criterion (4) to work well, the benefit functions of each core should accurately approximate the expected future core instruction rate starting from any given state  $s$ . In order to achieve this, we will tune the cost function parameters  $p_n$  using Reinforcement Learning (RL). The RL process basically learns to correlate the states  $s_t$  (as specified by different combinations of the state variables) observed on the core with the values of the reinforcement signal following these states. For the form of the benefit function we use, the RL updating process becomes:

$$p^i(t+1) = p^i(t) + \alpha_t[r_t + \gamma B^i(s_{t+1}, p^i(t)) - B^i(s_t, p^i(t))]\varphi(s_t), \quad (5)$$

where  $p^i(t)$  is the vector of parameters for core  $i$  at time  $t$ ,  $\alpha_t$  is a decreasing learning rate that is usually set to  $\alpha_t=1/t$ ,  $r_t$  is the reinforcement signal received at time  $t$  (observed system-wide instruction rate between times  $t$  and  $t+1$ ),  $\gamma$  is a discounting factor between 0 and 1 (value of 0.9 usually works well in practice), and  $\varphi(s)$  is a vector of all basis functions  $\varphi_n(s)$  used in defining the cost function  $B$ . The above equation has been proven to converge to the optimal parameter vector  $p^\infty$  such that  $B(s, p^\infty)$  provides the best approximation to the expected discounted sum of future reinforcement signals [7].

In order to see the intuition behind equation (5), notice that if the parameter vector  $p(t)$  is such that  $B(s, p_t) < B(s, p_\infty)$ , then on average we will observe  $r_t + \gamma B(s_{t+1}, p_t) > B(s_t, p_t)$ , which will lead to  $p(t+1) > p(t)$  after executing the update in (5), hence increasing the value of  $B(s_t, p)$  and making it closer to the true value  $B(s_t, p_\infty)$ . A similar logic applies in the reverse case if  $B(s, p_t) > B(s, p_\infty)$ .

As the accuracy of the benefit function increases in the course of RL, the system will be able to follow

more and more accurately the scheduling policy maximizing our target metrics.

Our plans for future work include implementing and evaluating this algorithm on a HMC system.

## 6. Related Work

In a recent study Kumar et al. demonstrated that assigning threads to the core most suitable for each thread improves performance on HMC systems [5]. Our work builds on this study by showing how a scheduler can combine a performance maximizing objective with objectives of balanced core assignment and response time fairness.

In another study, Kumar et al. demonstrated that heterogeneity conscious thread-to-core assignment can save power [4]. We believe that it will be possible to configure our scheduler to balance performance and power savings by incorporating power consumption as another objective into our scheduling algorithm, similarly to how we incorporated response time fairness and core assignment balance.

A study by Balakrishnan et al. demonstrated that applications suffer from jittery performance on HMC systems and proposed a fix to the scheduler that eliminates jitter in situations when system load is light [2]. The experiments described in our paper confirm the results of that study with respect to performance jitter. We improve on that study by (1) showing that jittery runtimes lead to inconsistent priority enforcement, and (2) presenting a fix to the scheduler that eliminates performance jitter regardless of system load.

Another area of related work is self-tuning algorithms based on reinforcement learning. One such algorithm was used in the past to guide memory and CPUs allocation on multi-partition multi-processor systems [8]. Another similar algorithm was used to tune file migration policies in multi-tier storage system [10]. Yet another RL based algorithm was used to solve a scheduling problem for soft real-time systems, where the goal is to maximize the number of jobs completing close to their deadlines [9]. Our RL based solution uses techniques and frameworks similar to those used in these previous algorithms. All of these referenced algorithms were

shown to bring substantial performance improvements, both in simulated [9,10] and real settings [8]. Therefore, we believe that an RL approach may be appropriate for our problem and that overheads will be sufficiently low to justify the deployment of our proposed RL algorithm in a commercial operating system.

## 7. Summary

We made a case that a scheduler on HMC systems must balance between three objectives: optimal performance, balanced core assignment and response time fairness. We demonstrated negative effects resulting from unbalanced core assignment: we confirmed a previous result showing running time jitter and presented a new result showing inconsistent priority enforcement. We presented a simple fix to the Linux scheduler that eliminates these negative performance effects by enforcing balanced core assignment. Finally, we presented an outline for a self-tuning scheduling algorithm that maximizes system-wide instruction throughput while maintaining an acceptable level of core assignment balance and response time fairness. In our future work we plan to implement this algorithm in a commercial operating system and evaluate its effectiveness.

## 8. References

- [1] SPEC CPU2000 web site. <http://www.spec.org>
- [2] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. In *Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)*, 2005
- [3] Bovet, D. and Cesati, M. Understanding the Linux Kernel, Chapter 10: Process Scheduling. *O'Reilly*, 2000
- [4] R. Kumar, K. Farkas, N. Jouppi, R. Parthasarathy, and Dean M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, 2003
- [5] R. Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, N. Jouppi, and K. Farkas. Single-ISA Heterogeneous Multicore Architectures for Multithreaded Workload Performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004
- [6] Richard McDougall and Jim Mauro. Solaris™ Internals: Solaris 10 and OpenSolaris Kernel Architecture. *Prentice Hall*, 2006
- [7] J. N. Tsitsiklis and B. Van Roy. An Analysis of Temporal-Difference Learning with Function Approximation. *IEEE Transactions on Automatic Control*, 45(5):674-690, May 1997
- [8] D. Vengerov. A Reinforcement Learning Approach to Dynamic Resource Allocation. *Engineering Applications of Artificial Intelligence*, 20(3):383-390, 2007
- [9] D. Vengerov. A Reinforcement Learning Framework for Utility-Based Scheduling in Resource-Constrained Systems. *Sun Microsystems TR-2005-141*, 2007
- [10] D. Vengerov. A Reinforcement Learning Framework for Online Data Migration in Hierarchical Storage Systems. *Journal of Supercomputing (to appear)*, 2007
- [11] A. Wierman and M. Harchol-Balter. Classifying Scheduling Policies with Respect to Unfairness in an M/GI/1. In *Proceedings of the SIGMETRICS*, 2003