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 thread schedulers for heterogeneous multicore systems should balance between three objectives: optimal performance, fair CPU sharing, and balanced core assignment. Thread-to-core assignment algorithms that optimize performance have been proposed in the past; in this paper we argue that they may conflict with the enforcement of fair CPU sharing. We also demonstrate the need for balanced core assignment: an unbalanced core assignment results in completion time jitter and inconsistent priority enforcement. We present a blueprint for a scheduling framework that balances between these three objectives. The framework consists of three components: performance-optimizing component, fairness component, and core balance component. While the performance-optimizing component could implement any previously proposed performance-optimizing algorithm, we propose a new algorithm based on reinforcement learning (RL). Our RL algorithm takes into account cache affinity, unlike previously proposed algorithms. Implementing and evaluating this framework and the RL algorithm is the subject of future work.

1. Introduction

A key objective pursued by thread schedulers in modern operating systems is to enforce fair CPU sharing. Although fairness policies vary from one scheduler to another, in general these policies minimize context switching, ensure that tasks are not starved for CPU time, and provide consistent response to users [4,7]. The advent of heterogeneous multicore (HMC) processors motivates new objectives and designs for scheduling algorithms. On a HMC system, optimal performance is achieved only by assigning a thread to run on the core that is best suited for it – i.e., the thread's *preferred* core [3,6]. Therefore, a scheduler must employ a core assignment policy that leads to optimal performance.

Unfortunately, the core assignment policy leading to optimal performance will conflict with the core assignment policy leading to fair CPU sharing. Consider the following example: Suppose that at the time when the scheduler dispatches thread *A* on CPU, *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 rate for *A*. Making the right decision requires carefully weighing performance/fairness tradeoffs of a given core assignment and making the choice that meets system's goals with respect to both objectives.

In some scenarios, 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. To eliminate performance jitter, a scheduler must periodically rotate threads among cores, thereby enforcing balanced core assignment. Clearly, such core assignment policy will not always result in threads running on their preferred cores and will thus conflict with the objective of optimizing performance.

All in all, the three objectives crucial to achieving good system performance and robustness, namely (1) optimal performance, (2) fair CPU sharing, and (3) balanced core assignment, conflict with each

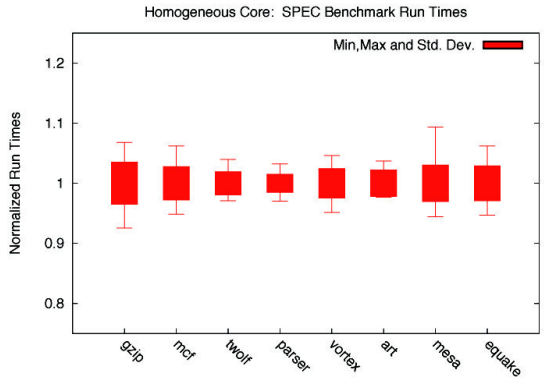

Figure 1. SPEC benchmark runtimes in the homogeneous setup.

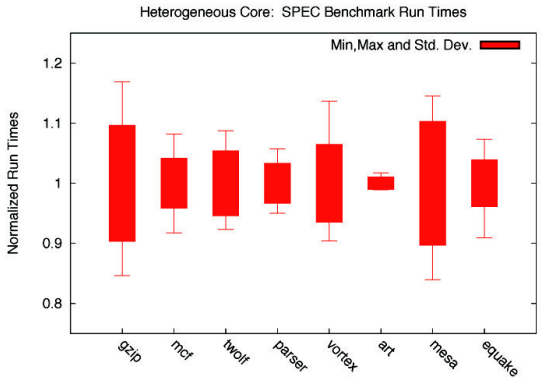

Figure 2. SPEC benchmark runtimes in the heterogeneous setup.

other. To resolve this conflict, the scheduler must ensure that the core assignment algorithms used to achieve these objectives take into consideration each other's core assignment decisions. This requires reconsidering the design of the operating system scheduler. In this paper we describe how we plan to address this problem.

The contributions of this paper are as follows:

(1) We show 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 eliminates jitter by enforcing balanced core assignment;
(3) We present a formal definition of a HMC scheduling problem in terms of balancing three objectives: optimal performance, fair CPU sharing, and balanced core assignment;
(4) We present a new scheduling framework that balances between the three objectives. The framework consists of three components: performance-optimizing component, fairness component, and core balance component. We also present a new algorithm for the performance-optimizing component based on reinforcement learning. Implementing and evaluating the framework and the algorithm is the subject of 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 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,

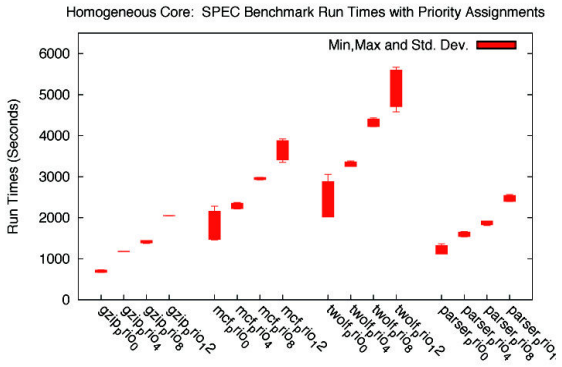

Figure 3. SPEC benchmark runtimes for varying priorities in the homogeneous setup.

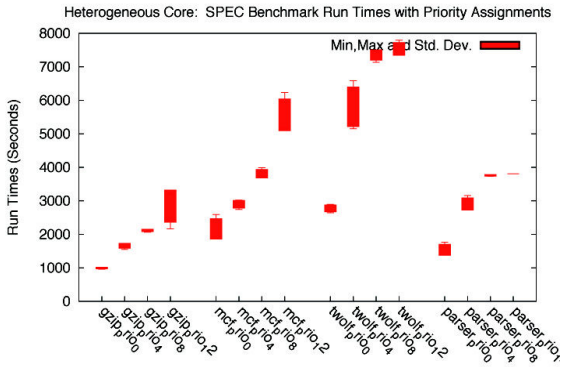

Figure 4. SPEC benchmark runtimes for varying priorities in the heterogeneous setup.

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; 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, 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.

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 core balancing algroithm 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 core balancing 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

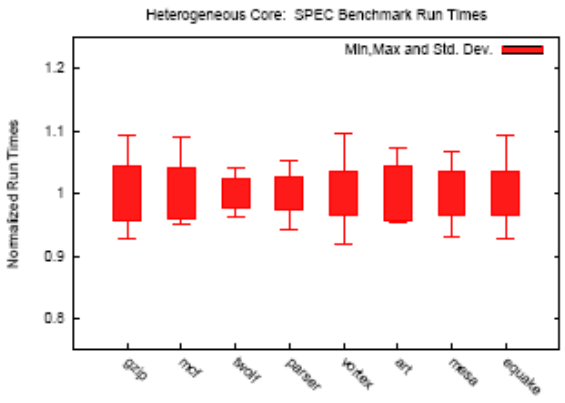

Figure 5. SPEC benchmark runtimes for varying priorities on the heterogeneous setup with the new scheduler.

`'hcs_balance_enabled'`, and an entry to `procfs` 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]; however it only worked in scenarios when the number of threads was smaller than the number of cores. Our core balancing algorithm works regardless ratio of threads to cores.

Our core balancing algorithm 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 new algorithm fixed runtime jitter, it did not consider how balancing threads' core

assignments would affect performance. Our algorithm frequently migrates threads among cores, which would interfere with performance-maximizing core assignment algorithms that attempt to run threads on cores that are best suited for them [3,6]. Our scheduling framework is meant to resolve this conflict.

4. The Scheduling Problem

A scheduler for HMC systems must balance between three objectives:

(1) Optimal performance
(2) Fair CPU sharing
(3) Balanced core assignment.

We recognize that *how* to balance between these objectives, i.e., the relative importance given to each objective, depends on particular circumstances. Our balancing framework presented in Section 5 gives the system administrator a flexibility to specify the importance of each objective, by setting the configuration parameters in the framework.

In this section we define the metrics for these objectives and formally define the scheduling problem on HMC systems.

4.1. Optimal performance

We measure performance in terms of the *aggregate normalized instruction per cycle rate* (IPC) achieved by all running threads. Normalized IPC (nIPC) for a thread is computed as weighted speed-up over the worst-suited core:

$$nIPC_{J,i} = \frac{IPC_{J,i}}{min\ (IPC_{J,k})}, \qquad (1)$$

where $IPC_{J,i}$ and $nIPC_{J,i}$ is the IPC and nIPC of thread J on core i, and min($IPC_{J,k}$) is the smallest IPC achieved by J across all system cores.

So the aggregate normalized IPC is the sum of all threads' performance improvements over the worst-suited core for each thread. Using the normalized IPC, as opposed to the raw IPC, eliminates the tendency to favour high-IPC threads [6].

4.2. Fair CPU sharing

As a metric for fairness when sharing the CPU, we use *Response Time Fairness* (RTF). A similar metric has been used in the previous work that evaluated fairness of scheduling algorithms [12]. The RTF metric is based on a *job's slowdown.* We define *job* as the portion of the thread run during a particular CPU timeslice. (The entire program can be thought of as a collection of jobs.) The job's *size* is simply the length of its CPU timeslice. Slowdown of a job of size x, $S(x)$, is defined as the job's response time $T(x)$ divided by its size:

$$S(x) = \frac{T(x)}{x}. \qquad (2)$$

$T(x)$ is the sum of the job's size and its queuing delay (i.e., the time spent waiting for CPU from the point of being placed on the scheduling queue).

DEFINITION 1: *A job is treated fairly if its expected slowdown is within a pre-defined constraint F:* $E[S(x)] \le F$.

This definition of fairness agrees with the *time-sharing* fairness policy used in Solaris™ and Linux [4,7]: larger jobs (i.e., threads with longer timeslices) will experience longer queuing delays than smaller jobs (i.e., threads with shorter timeslices).

Constraint F should be proportional to system load. We will determine a good setting for F experimentally. Since our goal is to design a scheduler that accomplishes the level of fairness similar to existing schedulers, we will measure typical bounds on slowdown achieved by existing schedulers, and will derive F based on that data.

DEFINITION 2: *System-wide response-time fairness (RTF) is the percent of jobs satisfying the fairness constraint in Definition 1.*

4.3. Core assignment balance

The metric for core assignment balance must capture the distribution of a thread's running time across the cores. For a thread J, we measure the fraction of time that J spent running on each core; then we measure the standard deviation from the average fraction. This standard deviation, CAB_J, is our metric for core assignment balance.

DEFINITION 3: *Core assignment for a thread J is considered balanced if:* $CAB_J \leq \varepsilon$. *(A setting for* ε *is determined experimentally.)*

DEFINITION 4: *System-wide core assignment balance (CAB) is the percent of tasks satisfying the balance constraint in Definition 3.*

A scheduler configured with a tight CAB constraint ε 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 new algorithm presented in the next section.

PROBLEM FORMULATION:

The goal of the scheduler on HMC systems is to maximize the aggregate normalized IPC over some time period t given constraints F for response time fairness and ε for core assignment balance, set *a priori* by the system administrator.

5. Scheduling Framework and Algorithm

We propose a scheduling framework that balances between the three objectives presented in the previous section. The framework consists of three components: performance-optimizing component, fairness component and core balance component. In addition, there is a master that co-ordinates these components.

Each component implements a core assignment algorithm that pursues that component's objective. For example, a performance-optimizing component could implement one of the algorithms proposed in previous work [3,6]. Or it could implement a new algorithm; we present a new algorithm for the performance-optimizing component in this section.

The fairness component implements a core assignment algorithm employed by existing schedulers to achieve fair distribution of CPU time: for example, a time-sharing algorithm used by default in Solaris and Linux.

The core balance component implements the balancing algorithm presented in Section 3.

Each of these components periodically decides that a particular thread should be *migrated* from the core where it executed in the past to another core. (Performance-optimizing components usually migrate threads at periodic intervals on the order of tens or hundreds of milliseconds. Fairness component and core balance component decide whether to migrate a thread upon expiration of the thread's timeslice.) In our framework, whenever any of these components decides to migrate a thread, it consults the master as to whether the migration is allowed. The master makes a decision based on the following relation:

$$B_{i,k-}+B_{j,k+} > B_{i,0}+B_{j,0}+a\cdot D_{CAB}+b\cdot D_{RTF} . \quad (3)$$

In this relation, $B_{i,0}$ and $B_{j,0}$ is the estimated normalized IPCs on cores i and j respectively if the thread is *not* migrated from i to j. $B_{i,k-}$ and $B_{j,k+}$ are estimated normalized IPCs on cores i and j respectively if the thread *is* migrated from core i to core j. D_{RTF} is the expected change in the system-wide RTF measure (per Definition 2) and D_{CAB} is the expected change in the system-wide CAB measure (per Definition 4) that will occur as a result of this migration. a, b are coefficients that show the relative importance of these three objectives and that are set *a priori* by a system administrator. The thread is migrated from core i to core j if the constraints in equation (3) are satisfied.

The migration criterion (3) implies that a particular thread migration is performed if the performance 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.

By using this design for our framework we assume that B_i, the normalized IPC on core i given the threads assigned to that core, can be estimated. This is a reasonable assumption: performance-optimizing core assignment algorithms presented in the literature already rely on such estimates [3,6]. Later in this section we will show how such estimate is obtained in the new performance-optimizing algorithm proposed in this paper.

This framework gives the system administrator a flexibility to specify the relative importance of each performance objective. For example, a system

administrator might decide that runtime jitter and inconsistent priority enforcement are not a concern for the system; in that case s/he can give an arbitrary low weight to the CAB metric (coefficient *a*), including completely ignoring it (setting *a* to zero).

Our future work involves implementing this framework with two different implementations of the performance-optimizing algorithm, one presented in previous work [3], and the new one, presented later in this section. We also plan to implement the framework in two different operating systems, Solaris and Linux, to make sure that it works with different fairness components. Another objective will be to find good settings for coefficients *a* and *b,* as these coefficients determine how the system balances between the three objectives.

5.1. A new performance-optimizing algorithm

We present a new self-tuning algorithm that assigns threads to cores on HMC systems in a way that maximizes normalized aggregate IPC. The algorithm is based on *reinforcement learning* (RL). It allows the system to *learn* the performance-optimizing core assignment policy. Learning is required for finding the optimal assignment 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 the *function* B_i that approximates the expected future normalized IPC on that core. Recall that the function B_i is used to make migration decisions (2).

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 normalized IPC on that core evaluated over the time between two consecutive migration decisions. The objective of parameter tuning is to make B_i a better approximation to the expected future value of the core IPC.

We now explain what variables are used to estimate B_i (Section 5.1.1), and how to update tunable parameters, so that B_i is progressively made to be a better estimate of the core IPC (Section 5.1.2).

5.1.1. Computing B_i

The function B_i uses several variables as inputs, and the first one is the ***average normalized IPC*** *(avgNIPC)* of threads on core *i*.

Another variable 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 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 function B_i will have the following form:

$$B(s_t, p) = \sum_{n=1}^{N} p_n \varphi_n(s_t), \qquad (4)$$

where $\varphi_n(s)$ are pre-specified *basis functions* defined on the space of possible values of *s,* and p_n, n=1, ..., *N,* are the tunable parameters that are adjusted in the course of learning. Basis functions can be specified using the framework presented in other RL algorithms used for resource allocation [9,10,11].

5.1.2. Updating tunable parameters

In order for the migration criterion (2) to work well, the B_i functions of each core should accurately

approximate the expected future core IPC starting from any given state *s*. The expected future IPC of the core can be defined as the expected discounted sum of future IPC observations. Then, one can tune the cost function parameters p_n using Reinforcement Learning (RL) in order to improve the approximation accuracy of functions B_i. 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, α_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 normalized IPC between times t and t+1), γ 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^∞ such that $B(s,p^\infty)$ provides the best approximation to the expected discounted sum of future reinforcement signals [8].

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, based on the definition of B_i as a discounted sum of future IPC observations, on average we will observe that $r_t + \gamma B(s_{t+1},p_t) > B(s_t,p_t)$, 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 B_i function increases in the course of RL, the system will be able to follow more and more accurately the core assignment policy maximizing performance.

6. Related Work

In a recent study, Kumar et al. presented a core assignment algorithm for maximizing performance on HMC systems [6]. Their algorithm considered only a performance maximization objective; it is an algorithm that could be used in the performance component of our scheduling framework, but by itself it does not balance performance with other objectives. Additionally, this algorithm was not implemented in a real operating system: it was evaluated via simulation. Kumar's algorithm used normalized IPC as a heuristic for core assignments. The authors found that the heuristic was a better predictor of a good core assignment if it was computed using multiple IPC samples; we will keep that in mind when developing our algorithm. Although our new RL algorithm also uses normalized IPC, it considers other factors, such as cache affinity, and learns the nature of interactions between the factors via reinforcement learning.

Becchi et al. presented a core assignment algorithm that relied on relative IPCs as a heuristic for deriving the optimal core assignment (similarly to Kumar's algorithm and to our RL algorithm) [3]. Like Kumar's algorithm, Becchi's algorithm focused on performance, and did not consider other objectives. Also, Becchi's algorithm has not been implemented in a real operating system. We will implement Becchi's algorithm in the performance component of our scheduling framework and will use it for evaluation of our scheduling framework and for comparison with our RL algorithm.

Balakrishnan et al. observed that heterogeneity of cores' computational power in HMC processors causes jittery runtimes [2]. We confirmed that results and showed that jittery runtimes lead to inconsistent priority enforcement. Balakrishnan suggested a simple change to the operating system scheduler to eliminate jitter; however it only worked in scenarios when the number of threads was smaller than the number of cores. We presented a core balancing algorithm that eliminates jitter even for workloads where the number of threads equals or exceeds the number of cores [5].

Another area of related work is self-tuning algorithms based on reinforcement learning. These algorithms were used to solve problems similar to

ours, such as guiding memory and CPUs allocation on multi-processor systems [9], tuning file migration policies in multi-tier storage system [10], and building a soft real-time scheduler, where the goal was to maximize the number of jobs meeting their deadlines [11]. These algorithms were shown to bring substantial performance improvements, both in simulated [10,11] and real settings [9]. To determine whether RL is applicable to our problem, we will compare it to simpler heuristic algorithms in terms of performance and runtime overhead.

7. Summary

We made a case that a scheduler on HMC systems must balance between three objectives: optimal performance, fair CPU sharing, and balanced core assignment. We showed that unbalanced core assignment results in performance jitter and inconsistent priority enforcement, and presented a simple fix to the Linux scheduler that eliminates jitter. Finally, we presented a scheduling framework that balances the three objectives, and proposed a new performance-optimizing core assignment algorithm based on reinforcement learning. The new algorithm, unlike previous similar algorithms, accounts for cache affinity. Implementing and evaluating the framework and the algorithm is the subject of future work.

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] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *Proceedings of the Conference on Computing Frontiers,* 2006

[4] Bovet, D. and Cesati, M. Understanding the Linux Kernel, Chapter 10: Process Scheduling. *O'Reilly*, 2000

[5] Alexandra Fedorova, D. Vengerov, and Daniel Doucette. Operating System Scheduling On Heterogeneous Multicore Systems. In *Proceedings of the the PACT'07 Workshop on Operating System Support for Heterogeneous Multicore Architectures,* 2007

[6] 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

[7] Richard McDougall and Jim Mauro. Solaris™ Internals: Solaris 10 and OpenSolaris Kernel Architecture. *Prentice Hall*, 2006

[8] 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

[9] D. Vengerov. A Reinforcement Learning Approach to Dynamic Resource Allocation. *Engineering Applications of Artificial Intelligence*, 20(3):383-390, 2007

[10] D. Vengerov. A Reinforcement Learning Framework for Online Data Migration in Hierarchical Storage Systems. *Journal of Supercomputing (to appear)*, 2007

[11] D. Vengerov. A Reinforcement Learning Framework for Utility-Based Scheduling in Resource-Constrained Systems. *Sun Microsystems TR-2005-141,* 2007

[12] A. Wierman and M. Harchol-Balter. Classifying Scheduling Policies with Respect to Unfairness in an M/GI/1. In *Proceedings of the SIGMETRICS,* 2003