Age Based Scheduling for Asymmetric Multiprocessors

Nagesh B. Lakshminarayana, Jaekyu Lee, Hyesoon Kim
School of Computer Science
Georgia Institute of Technology
{nageshbl, jaekyu.lee, hyesoon}@cc.gatech.edu

ABSTRACT

Asymmetric (or Heterogeneous) Multiprocessors are becoming popular in the current era of multi-cores due to their power efficiency and potential performance and energy efficiency. However, scheduling of multithreaded applications in Asymmetric Multiprocessors is still a challenging problem. Scheduling algorithms for Asymmetric Multiprocessors must not only be aware of asymmetry in processor performance, but have to consider the characteristics of application threads also.

In this paper, we propose a new scheduling policy, Age based scheduling, that assigns a thread with a larger remaining execution time to a fast core. Age based scheduling predicts the remaining execution time of threads based on their age, i.e., when the threads were created. These predictions are based on the insight that most threads that are created together tend to have similar execution durations. Using Age based scheduling, we improve the overall performance of several important multithreaded applications including Parsec and asymmetric benchmarks from Splash-II and OmpSCR. Our evaluations show that Age based scheduling improves performance up to 37% compared to the state-of-the-art Asymmetric Multiprocessor scheduling policy and on average by 10.4% for the Parsec benchmarks. Our results also show that the Age based scheduling policy with profiling improves the average performance by 13.2% for the Parsec benchmarks.

Categories and Subject Descriptors

D.4.1 [**Operating Systems**]: Process Management — Scheduling; C.1.3 [**Processor Architectures**]: Other Architecture Styles — Heterogeneous (hybrid) systems

General Terms

Performance

Keywords

Thread Scheduling, Asymmetric Multiprocessors, Age Based Scheduling

SC09 November 14-20, 2009, Portland, Oregon, USA (c) 2009 ACM 978-1-60558-744-8/09/11... $10.00.

1. INTRODUCTION

Multiprocessors are becoming the main stream of computing platforms and heterogeneous architectures represent an increasingly popular class of multiprocessors. Heterogeneous architectures typically include one or more specialized cores or accelerators together with the main CPU(s). The specialized cores or accelerators help the CPU to perform certain computations several times faster than the CPU could have on its own. Asymmetric Multiprocessors (AMPs) represent a new kind of heterogeneous architectures. These architectures include CPUs of unequal performance. Usually, all the CPUs have the same ISA. Asymmetric architectures can provide significantly better performance than conventional, i.e., symmetric architectures which occupy the same die area and consume the same amount of power [20] [16]. These architectures generally include few fast cores and several slow cores. They are meant to provide power-performance effective platforms for both throughput-oriented applications and applications that need good serial performance. However, asymmetric architectures present new challenges to the operating system community, which until now has assumed that the hardware underneath the operating system is homogeneous. One of these challenges is scheduling of threads on the available processors.

Thread scheduling is one of the most important and fundamental services offered by an operating system kernel. Some of the metrics an operating system scheduler seeks to optimize are: fairness, throughput, turnaround time, response time and efficiency [22]. In an AMP environment, many of the assumptions based on which the traditional scheduling policies for multicores/multiprocessors[1] are designed become untrue. Multiprocessor operating systems assume that all cores are identical and offer the same performance. In a Symmetric Multiprocessor, since all cores are identical, the scheduler makes scheduling decisions based solely on the load on each core. But, in the case of AMPs, not only the loads on the individual cores, but their relative computational power should also be considered. Balakrishnan et al. [6] showed that having an asymmetry unaware scheduler would not only result in bad application performance, but can also cause application instability.

Figure 1 shows the performance of the Parsec benchmark suite [8] for three different machine configurations, namely, all-fast, half-half and all-slow. All-fast and all-slow are symmetric configurations, where as half-half is an asymmetric configuration. These configurations are explained in greater detail in section 6. The scheduler used for this experiment is the Linux Kernel 2.6.18 scheduler [18], which is not aware of the asymmetry of processors. Hence, even though half-half has more computing power than all-slow, some applications such as blackscholes, fluidanimate and swaptions perform on half-half as slow as they do on all-slow. This

[1]the terms core and processor will be used interchangeably

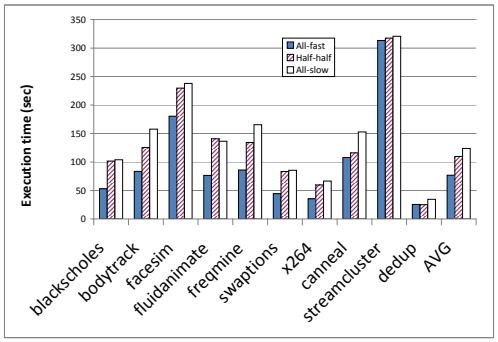


Figure 1: Experiments in the three machine configurations (Parsec)

is because threads on slow cores become critical threads which increase the execution time of the entire application. Thus, for AMPs we need a scheduler that can detect critical threads and boost their performance.

We propose a scheduler that can predict the thread with the longest remaining execution time. We predict the remaining execution time of threads based on their relative age i.e., when the threads were created. Predicting the exact lengths of threads or jobs at run-time is extremely challenging. Instead of predicting the actual lengths of threads, our mechanism predicts relative execution durations. We found that threads that are created around the same time tend to have similar execution durations. We can find examples of this in parallel-for structures or applications with fork-join models. The main reasons for this similarity across threads are: (1) programmers tend to write one piece of code and increase the number of threads that execute the same code (2) programmers tend to do static load balancing between threads to utilize all cores. Therefore, though we might not be able to predict how long each thread will execute, we can easily predict which thread within a thread group (explained in section 3.2) is lagging behind, because all threads in the same group tend to have similar execution lengths. The thread that is lagging behind can then be made to execute on a fast core. Scheduling based on the relative age of threads is called *age based scheduling*.

Operating system scheduling for multiprocessor systems is a well researched topic. However, the schedulers proposed for multiprocessors are inadequate for asymmetric architectures. Schedulers of well known operating systems such as Linux [9] [18] and FreeBSD [19] focus on providing fairness and good response time to interactive applications; they are not designed and implemented for asymmetric architectures. Recent asymmetry aware algorithms [14, 5, 13, 17] will be discussed in Section 8.

The contributions of our paper are as follows:

1. We propose a simple, but very effective relative thread length prediction mechanism.
2. We propose a new asymmetry aware thread scheduling policy, which considers the relative length of each thread.
3. We thoroughly analyze a wide range of workloads to provide an insight into symmetry and asymmetry of workloads.

2. BACKGROUND AND PROBLEMS WITH CURRENT SCHEDULING ALGORITHMS

In this section, we provide background knowledge about existing scheduling policies in desktop operating systems using the Linux scheduler [3, 9, 18] as reference and explain why existing schedulers are not suitable for AMPs. Note that our discussions assume that all threads have the same priority (or weight).

In a traditional multiprocessor operating system, the scheduler tries to provide fairness to the threads in the system. Towards this goal, the scheduler tries to keep the threads distributed equally among the available cores. This ensures that all threads get equal opportunities to execute and also improves the system throughput since all cores are utilized efficiently. The goal of the current Linux scheduler is the fair and optimal use of the available CPU resources.

2.1 Thread Assignment

In keeping with the fairness and optimal use goals mentioned earlier, in current operating systems, the decision of where to assign a thread is made solely based on the load (number of threads assigned or sum of weights of threads assigned) on each core. Whenever a new thread is created, it is assigned to the least loaded core, i.e., the core that has the fewest threads in its run queue. A similar policy is used to assign threads that have woken up. This policy works well on a machine with all identical cores, but does not provide good results in an asymmetric environment [6]. Suppose that a machine has two cores - one fast core running at 2.5 GHz and a slow core running at 500 MHz. An application with four threads is to be run on the machine. In such a situation, how should the assignment of threads to cores be done? current operating system schedulers would assign two threads to each core and let the threads run on the initially assigned cores till their termination. This would result in bad application performance since the fast core would remain idle once the threads assigned to it terminate and, all threads would not be able to take advantage of the fast core.

2.2 Load Balancing and Idle Balancing

Multiprocessor operating systems do load balancing [3] periodically to ensure that uniform distribution of load across the cores is maintained. Creation of new threads, termination of threads, blocking and unblocking of threads result in load imbalance across cores, so periodic load balancing is required. Load balancing is different from thread context switching and is performed less frequently than context switching. Idle balancing is needed when a core goes idle, i.e., all the threads that were assigned to the core have either terminated or blocked and the core's run queue is empty. In case of idle or load balancing, threads are pulled from a busy core to the idle or lightly loaded core. In current operating systems, load balancing and idle balancing consider only the number of threads (or load calculated using per thread weights) and result in the same problem as asymmetry unaware thread assignment. In AMPs, even load balancing and idle balancing should be done in an asymmetry aware fashion to make optimal use of cores. Often, it may be required to migrate threads from slow cores to fast cores to ensure that fast cores do not remain idle.

3. SCHEDULING IN AMPS - AGE BASED SCHEDULING

3.1 Overview

Age based scheduling assumes that the majority of multithreaded applications follow a simple fork-join model as shown in Figure 2.[2] The main thread forks several child threads and then could block immediately waiting for the child threads to complete or could block after performing some work. The application threads may encounter several barriers during their execution. The goal of Age

[2]Some applications do not follow this model

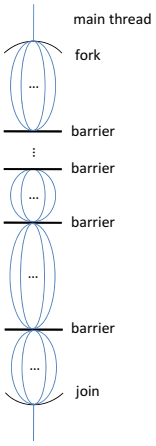


Figure 2: Application model assumed by age based scheduling

based scheduling is to schedule threads such that they all reach their next milestones, which could be either a barrier or termination, at the same time. It tries to do this by utilizing all the available cores, while exploiting the fast cores to accelerate threads that are lagging behind. Because of the asymmetric nature of the cores, threads that execute on slow cores automatically lag behind threads executing on fast cores.

Figure 3 shows an example of fork-join model in one of the Parsec benchmarks, blackscholes. The structure of blackscholes is identical to the application model explained above. The main thread creates several child threads (pthread_create()) and blocks on the child threads (pthread_join()), waiting for them to terminate. Each of the child threads first reaches a common barrier (pthread_barrier_wait()), after crossing the barrier each child thread executes till it terminates. After all the child threads have terminated, the main thread continues its execution and terminates. Swaptions, also from Parsec, has the same code structure as well, except that it contains no barriers.

```
/* function executed by each child thread */
int bs_thread(void *tid_ptr) {
    ...
    pthread_barrier_wait(&barrier);

    for (j = 0; j < NUM_RUNS; j++) {
        for (i = start; i < end; i++) {
            price = BlkSchlsEqEuroNoDiv(...);
            ...
        }
    }
    return 0;
}
/* main */
int main (int argc, char **argv) {
    ...

    /* fork */
    for (i = 0; i < nThreads; i++) {
        tids[i] = i;
        pthread_create(&_M4_threadsTable[i], NULL,
                       bs_thread, &tids[i]);
    }

    /* join */
    for (i = 0; i < nThreads; i++) {
        pthread_join(_M4_threadsTable[i], &_M4_ret);
    }
    ...
}
```

Figure 3: Skeleton code of Blackscholes

3.2 Degree of Symmetry

The key component of Age based scheduling is that threads are symmetric, i.e., they have equal amount of work, if they are created at similar times. To better understand how many applications actually have symmetric behavior, we measure the degree of symmetry. First, we illustrate the concept of thread groups. Figure 4a shows 2 different thread groups, the main thread and all child threads. Figure 4b shows 4 thread groups. Threads that are created together form a thread group.

Figure 5 shows the degree of symmetry of the Parsec benchmarks (the degree of symmetry was computed using the simmedium

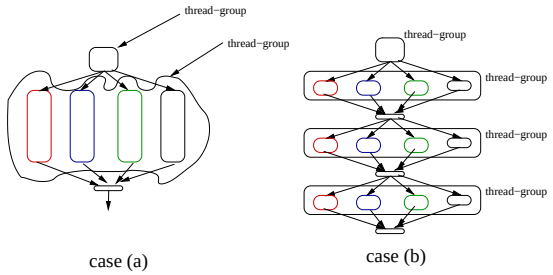


Figure 4: Thread-group examples

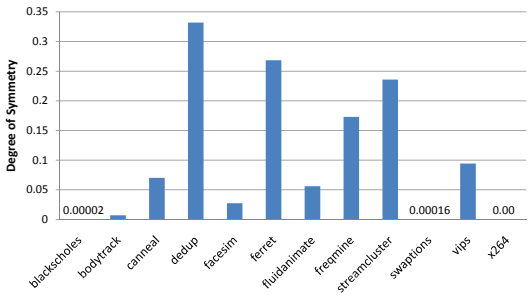


Figure 5: Degree of Symmetry of the Parsec benchmarks

input). The degree of symmetry indicates how symmetric the (child) threads of an application are with respect to each other. A value closer to zero indicates a high degree of symmetry, whereas a higher value indicates a lower degree of symmetry. We regard applications with degree value ≤ 0.1 as symmetric and all others as asymmetric. The degree of a thread group is calculated as standard deviation of the lengths of threads in the thread group divided by the average of the lengths of the threads in the thread group. If an application has more than one thread group, we then average the degree of symmetry of all the thread groups to obtain the final value for the application (note that the main thread is not used in the calculation of degree of symmetry). From Figure 5 we see that eight out of the twelve shown the Parsec benchmarks are symmetric.[3] This symmetry data forms the basis of our insight that threads that are created together tend to have similar execution times.

3.3 Details of the Policy

Based on the Age of threads, we propose the *Age Based Longest Job Fast Core First (LJFCF)* policy. LJFCF is similar to the *Longest Job First (LJF)* scheduling policy [22]. LJFCF predicts how far a thread is from the next barrier, or if no more barriers are going to be encountered, its termination. The thread(s) with the longest predicted distance(s) to the next milestone is(are) assigned to the fast core(s). Predicting the exact distance or execution duration to the next milestone is very difficult. However, for LJFCF, we do not have to know the absolute values of distances of different threads, we have to know only the relative values of distances of threads. The prediction of relative values of distances to the next milestone is based on the insight that threads that are created together usually have the same lengths. Thus, threads that are created together are predicted to have the same distances to their next milestones at the time of their creation and threads that are created later are predicted to have longer distances to their next milestones. When a thread reaches a milestone, its distance to the next milestone is predicted and the newly predicted distance is used for thread to core assignment. Note that Age based scheduling could cause more migrations

[3] x264 has a degree of symmetry of zero since it has only one thread in each thread group

than other mechanisms. However, our simulation results show that the benefit of Age based scheduling can overcome overhead from frequent migrations.[4]

LJFCF is applied whenever one of the following occurs: (1) a thread is created, (2) a core goes idle because all the threads assigned to the core have either terminated or blocked, or (3) periodic timer for reassignment expires. A thread may be blocked for several reasons - waiting for a lock, waiting for all threads to reach a barrier, waiting for a child to terminate, waiting for I/O to complete etc. For LJFCF, we have replaced periodic load balancing with periodic reassignment. While load balancing is typically done for each core at different points in time, reassignment is done at the same time for all cores. In our experiments, the reassignment interval was same as the load balancing interval used for policies that do load balancing. Reassignment is done periodically to ensure that threads that are lagging behind can catch up; this results in improved application performance. Algorithm 1 summarizes the LJFCF scheduling algorithm. How to calculate rem_exe used in Algorithm 1 will be explained in Section 3.4.

3.4 Age Prediction

There are three ways of calculating rem_exe: oracle, prediction, and profiling. Oracle method is used to demonstrate the effectiveness of scheduling policies. The following two sections describe the LJFCF policy based on prediction and profiling.

3.4.1 LJFCF Policy using Prediction

The LJFCF Policy using Prediction [Age(Pred)] uses prediction to determine the distance of each thread from its next milestone. Age(Pred) predicts that the distance between any two successive milestones (creation, barriers and termination) of any thread is the same and the common distance value is predicted to be very large. This implies that after crossing a milestone all threads are predicted to have the same distance to their next milestone. As a thread executes, the predicted remaining distance to the next milestone reduces according to its progress. When Age(Pred) is invoked, the predicted remaining distance to the next milestone is used as *rem_exe* in Algorithm 1 to assign threads to cores.

3.4.2 LJFCF Policy using Profiling

In Age(Pred), it is predicted that any two successive milestones of any thread have the same distance between them, but in LJFCF Policy using Profiling [Age(Prof)], it is predicted that the distance between any two successive milestones of only a *given* thread are the same. The distances between successive milestones are not the same across threads. By profiling the application with sample inputs, the average distance between milestones is determined for each thread. These average distances are fed to the scheduler as a set of ratios before the application is run. Based on these ratios, the remaining distance of each thread to its next milestone is predicted. Threads with lower ratios are predicted to have shorter distances between milestones. Thread assignment is done using Algorithm 1 using the predicted, ratio based remaining distances.

3.5 Why should Age Based Scheduling Work?

We show why Age based scheduling should work and provide good application performance by comparing with the state-of-the-art AMP scheduler algorithm, which was proposed by Li et al. [17] (referred to as Li's mechanism in this section). The key ideas of

[4]Our simulator models all the migration overhead faithfully and reduction in number of migrations is one of our future research work

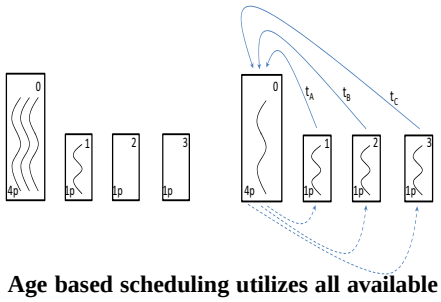


Figure 6: Age based scheduling utilizes all available resources (left - Li's mechanism, right - Age based scheduling)

Li's mechanism are Asymmetry-Aware Load Balancing and Faster-Core First Scheduling. Asymmetry-Aware Scheduling ensures that the load on each core is proportional to its computational power, this provides fairness to the threads of an application. Faster-Core First Scheduling assigns threads to fast cores if they are underutilized. We use a machine with four cores - one fast core (Core 0) and three identical slow cores (Core 1, Core 2 and Core 3) - as shown in Figure 6 to illustrate the working of Age based scheduling and Li's mechanism. The fast core has four times the performance of each slow core, this is indicated by the 4p inside the block representing the fast core in the figure.
Li's mechanism has the following limitations:

1. Using the concept of scaled load that is explained in section 5.1, Li's mechanism assigns the threads of a multithreaded application as shown in Figure 6left. This results in low utilization of the slow cores when there are few threads in the system. On the other hand, Age based scheduling assigns threads as shown in Figure 6right, resulting in high utilization of cores.
2. Li's mechanism does not exploit fast cores. Since a fast core is assigned threads proportional to its computational power, it is being treated as though it is composed of several slow cores and each *virtual* small core is assigned the same load as a *real* small core in the system. Also, once threads start executing, they continue to execute on the same core till they terminate, block or are migrated. This may be unfair to threads executing on the slow cores. Age based scheduling tries to exploit fast cores by giving all threads (or threads that are lagging behind) opportunities to accelerate by executing on fast cores. In Figure 6, with Li's mechanism, once initial assignment of threads is made, threads continue to execute on the same cores unless some event such as creation, termination or blocking of a thread occurs. In case of Age based scheduling, threads executing on slow cores are moved to fast cores at different points in time i.e., $t_A \neq t_B \neq t_C$ in Figure 6, so that all threads execute on the fast cores.

4. IMPLEMENTING LJFCF POLICY

4.1 Tracking the Progress of a Thread

For each thread, LJFCF predicts the remaining distance to its next milestone. As a thread executes, the predicted distance should reduce. Thus to predict the remaining distance we have to track the progress of threads accurately. We approximate the progress of a thread with the number of instructions executed by it. To count the number of instructions executed by each thread a hardware mechanism is needed. Today's processors already have a hardware performance counter *INST_COUNT* (Intel and AMD processors provide the performance events INST_RETIRED and RETIRED_INSTRUCTIONS respectively, to count the number of re-

Algorithm 1 Longest Job Fast Core First Policy (LJFCF)

STEP1
sort the threads in the decreasing order of their *rem_exe*
STEP2
if the number of threads is less than or equal to the number of cores **then**
  make 1:1 assignment, with threads with longer *rem_exe*s being assigned to fast cores
end if
STEP3
if the number of threads is greater than the number of cores **then**
  compute the average *rem_exe* that must be assigned to each core *i*
  $avg\_rem\_exe\_core_i = (rem\_exe_{total}/core\_perf_{total}) * core\_perf_i$
  for each thread $thread_j$ in sorted order **do**
    identify the core $core_k$ for which the difference between the *rem_exe* to be assigned and the *rem_exe* actually assigned is the highest
    $rem\_rem\_exe_i = avg\_rem\_exe\_core_i - rem\_exe\_core_i$
    $k = max(rem\_rem\_exe_i)$
    assign $thread_j$ to $core_k$
    if number of threads yet to be assigned is less than or equal to the number of cores without any assignments **then**
      make 1:1 assignment of threads to cores
      break
    end if
  end for
end if

tired instructions) to count the number of instructions executed [15]. The operating system can simply utilize one such existing hardware performance counter. To remember the progress made by a thread, a new field, *progress*, is added to the task/thread data structure. Operating systems maintain one task structure variable for each thread in the system. When a thread is created, the *progress* field in its task structure is initialized to zero. The operating system updates the *progress* field whenever the thread executes. When a thread is switched in and starts executing, *INST_COUNT* is reset to zero and incremented for each instruction that is completed. Whenever a hardware interrupt or exception occurs, *INST_COUNT* is disabled i.e., it stops counting and is enabled again when the interrupt or exception handler returns. *INST_COUNT* is not disabled when software interrupts are raised. This way, work done for a thread inside system calls is not ignored. At every timer tick, the operating system adds the value accumulated in *INST_COUNT* to *progress* and resets *INST_COUNT* to zero. *progress* is also updated with *INST_COUNT* when the thread blocks, is switched out or is even migrated. In this way, *progress* of each thread is tracked and the predicted remaining distance is kept updated.

4.2 Passing Profile Information with Compiler assistance

Applications are profiled during compilation to collection information about the execution of different application threads. The information collected during compilation is stored in a data segment in the binary. To store profile information of applications, the operating system scheduler implements *prof_data* data structure. When a binary is loaded, the data segment containing profile data is read and *prof_data* is updated with profile information. When an application thread is created, the scheduler reads *prof_data* for profile information of the application. If profile data is available, the scheduler updates the thread's task structure with the profile data. In case profile information is not available for all threads created by the application, the scheduler does extrapolation to determine the information for the newly created thread.

4.3 Passing Profile Information without Compiler assistance

To be able to pass profile information to the scheduler without any aid from the compiler we need a mechanism that allows users or user applications to communicate with the scheduler. Since *prof_data* is a kernel data structure, user programs cannot access it directly. So, an interface to update or read *prof_data* is provided through an entry in the proc file system. A user or a tool can write the application path, the desired action (add, modify, or delete) and any needed support data to the proc entry. Whenever the proc entry is written to, *prof_data* is updated or read from according to the specified action.

4.4 Implementation in Operating System

Essentially, Algorithm 1 has to be implemented in the operating system kernel. The following sections of the scheduler in the operating system have to be changed to implement the algorithm: (1) initial thread assignment, (2) thread termination, (3) assignment on thread wake up, and (4) load balancing. Since Algorithm 1 is centralized, the algorithm could always be executed on the same core. It could be the fastest core, or the core with the lowest operating system assigned id and so on.

4.5 Scalability

LJFCF has a complexity of O(nlogn + pn), where n is the number of threads and p is the number of cores. For a machine with a small number of cores the complexity of LJFCF becomes O(nlogn). When we have a large number of threads or a large number of cores, the algorithm can be extended in a hierarchical fashion. In Algorithm 1, the scheduler sorts all threads based on their remaining execution time. Instead of globally sorting all the threads, each core or a group of cores can sort threads locally first. Only the longest job from each core or group can be a candidate for global migration. The rest of the threads are scheduled locally.

5. OTHER SCHEDULING POLICIES

A complete scheduling mechanism specifies three main policies - policy for initial assignment, policy for wake up assignment and policy for load balancing. The roles of these policies were explained in section 2. Below, we briefly explain the different policies that were used for each category of policy in our evaluation of Age based scheduling.

5.1 Thread Assignment Policies

Asymmetry Unaware Assignment (AUA) [3, 9, 18]: This is the assignment policy used by current operating systems. It is not

aware of asymmetry of cores and it assigns threads to the core that has the least number of threads.

Fast Core First Assignment (FCA)*[5]: This policy assigns a thread to a fast core first if a fast core is idle. If not, it assigns a thread to the next idle core.

Asymmetry Aware Assignment (AAA) [17]: It defines what is called as scaled load *L* for each core. The performance of the slowest core is taken as 1 and the performance of a core that is *F* times faster than the slowest core is taken as *F * S*, where *S* is the scaling factor. For simplicity, it is taken to be a constant of value less than 1. The scaled load for each core is equal to its run queue length divided by its performance.

When a new thread is to be assigned a temporary scaled load is computed for each core assuming that the new thread has been assigned to the core. The thread is then assigned to the core with the lowest temporary scaled load. In the event that two or more cores have the lowest temporary scaled load, the thread is sent to the slowest of the tied cores. This assignment ensures that a new thread runs on a fast core if it is underutilized and also allows multiple threads to run on the fast cores.

5.2 Wake up Assignment Policies

These policies are used to assign newly woken up threads to cores. Threads can block waiting for a child to finish, waiting for a lock or waiting for some other events. When the event on which a thread is waiting, occurs, the thread is woken up and assigned to a core.

Previous Core (PW)*: When a thread is woken up, it is assigned to the core on which it was previously running. This policy tries to reduce the overhead due to thread migration. This policy can be effective if threads are woken up after a short wait and the previous core is not occupied by other threads. However, if the previous core is already running other threads, this policy can cause a load imbalance problem.

Faster Core (FW)*: When a thread is woken up, it is assigned to the least loaded fast core. This policy is trying to increase the utilization factor of fast cores. However, this can results in very severe load imbalance problems.

Idle Core (IW) [3, 9, 18]: The idle core (IW) policy is similar to the policy in the Linux scheduler. When a thread is woken up, it is assigned to an idle core, if one is available, else it is assigned to the previous core on which it was running. This policy is the best for solving load imbalance problem.

Suitable Core (SW)[our proposed mechanism]: This policy is the most sophisticated policy. The suitable core (SW) policy checks if any idle fast cores are available, if so, the thread is assigned to an idle fast core. When looking for idle fast cores preference is given to the core on which the thread was previously running if that core was a fast core. If no fast cores are available, a check for an idle slow core is made in the same way as the check for an idle fast core. If an idle slow core is found, the thread is assigned to the idle core. Preference is given to the core on which the thread was previously running because the initial overhead (cache misses) experienced by the thread will be low if it is assigned to the same core instead of a different core. The simulator models this effect as well.

5.3 Load Balancing Polices

Asymmetry Unaware Balancing (AUB) [3, 9, 18]: This is the load balancing used by current operating system schedulers. It is not aware of asymmetry of cores and it does balancing based on the number of threads assigned to each core (or based on the total weight of threads assigned to each core).

Round Robin Balancing (RRB)*: This policy migrates threads to a fast core in a round-robin fashion. This policy tries to provide fair access to the fast cores for the threads in the system.

Asymmetry Aware Balancing (AAB) [17]: The Load Balancing policy tries to ensure that the difference between the maximum scaled load L_{max} in the system and the minimum scaled load L_{min} in the system is less than or equal to 1 i.e., $L_{max} - L_{min} \leq 1$. This results in threads being assigned to cores in a fair manner.

5.4 Combining Policies

The policies explained above can be combined in different ways to obtain many different combinations. For the rest of the paper, we represent different scheduling policies obtained by combining one policy of each category by joining their names using hyphens. From left to right, the names are specified in the order - thread assignment policy, thread wake up policy, and load balancing policy. For example, AUA-IW-AUB means that AUA thread assignment policy, IW thread wake up policy and AUB load balancing policy. This is the current Linux scheduling policy and is called LIN in our discussions. The mechanism proposed by Li et al. is AAA-IW-AAB and will be referred to as SCALELD. Note that if an Age based policy is used as the third policy (load balancing policy) in a combination, it actually performs periodic reassignment as explained in 3.3 instead of periodic load balancing.

Table 1 summarizes the performance of the different policy combinations. (-) means low performance and (+) means high performance. The number mentioned for each policy combination in Table 1 specifies the reason for categorizing the performance of that policy combination as either (-) or (+). The list of reasons is given below. We found that wake up assignment policies do not affect the overall performance significantly compared to other polices, hence there are no separate entries for different wake up policies.

1. Since both initial assignment and load balancing polices are asymmetry unaware, the performance would be very low.
2. Since load balancing policy is asymmetry unaware, performance would be low.
3. Since the initial assignment is asymmetry unaware, performance could be low.
4. Asymmetry aware assignment and load balancing should provide good performance.
5. Since initial assignment is not intelligent, performance could be low.
6. Age based assignment and asymmetry aware load balancing should provide good performance.
7. Asymmetry unaware initial assignment and a simple round robin mechanism would result in low performance.
8. Simple round robin would result in low performance.
9. Asymmetry aware scheduling in combination with age based balancing should provide good performance.
10. Age based assignment and scheduling should provide good performance.

Certain combinations of policies in Table 1 will be referred to in the evaluation section using names shown in Table 2.

6. EXPERIMENTAL METHODOLOGY

6.1 Simulation Methodology

We use an in-house, cycle accurate simulator for our experiments. The simulator is a hybrid incorporating features of both

[5]* indicates a policy that might not be used in an operating system, but we discuss it just to complete all the possible basic policies

Table 1: Performance of different possible policies and the reason

Load Balancing Policy	AUA	AAA	FCA	Age(Oracle)	Age(Pred)	Age(Prof)
	PW/FW/IW/SW	PW/FW/IW/SW	PW/FW/IW/SW	PW/FW/IW/SW	PW/FW/IW/SW	PW/FW/IW/SW
AUB	1 (–)	2 (–)	2 (–)	2 (–)	2 (–)	2 (–)
AAB	3 (–)	4 (+)	5 (–)	6 (+)	6 (+)	6 (+)
RRB	7 (–)	8 (–)	8 (–)	8 (–)	8 (–)	8 (–)
Age(Oracle)	3 (–)	9 (+)	5 (–)	10 (+)	10 (+)	10 (+)
Age(Pred)	3 (–)	9 (+)	5 (–)	10 (+)	10 (+)	10 (+)
Age(Prof)	3 (–)	9 (+)	5 (–)	10 (+)	10 (+)	10 (+)

Table 2: Scheduling mechanisms

Combination	Name	Comment
AUA-IW-AUB	LIN	Scheduling policy in Linux
AAA-IW-AAB	SCALELD	Scheduling policy proposed by Li et al.
AUA-IW-RRB	RR	Threads are moved to fast cores in a Round-Robin manner
FCA-SW-Age(Pred)	FCA-AGE	Fast Core First assignment with Prediction based LJFCF
Age(Oracle)-SW-Age(Oracle)	AGE(ORACLE)	Oracle based LJFCF
Age(Pred)-SW-Age(Pred)	AGE	Prediction based LJFCF
Age(Prof)-SW-Age(Prof)	AGE(PROF)	Profiling based LJFCF

trace driven and execution driven simulators. The architecture simulations are done using the trace driven part, but all the operating system relevant events are precisely modeled using operating system algorithms through the execution driven part.

The simulator performs initial assignment, wake up assignment, load and idle balancing according to the specified parameters. It also performs time slicing between the threads in a ready queue. By default, a time slice of 10 ms is assumed. Even context switch and migration overheads are simulated.[6] All threads of an application are assumed to have the same static and dynamic priorities. Equal static priorities means that all threads are allocated the same time slice and because of equal dynamic priorities all threads in a give ready queue are executed in a round-robin fashion.

The traces for simulation are generated using a Pin tool. The traces provide information about each application thread in the form of sections. Each section can be either a critical section or non-critical section. In addition to the type of the section, the length of the section is also included. Information about the locks associated with critical sections and the various barriers and joins encountered by a thread are also included in the traces. The inclusion of synchronization information enables us to simulate thread interactions accurately.

6.2 Real Machine System

To verify our simulator, we compared the simulator's results with results from a real machine system. We use *SpeedStep* technology [4] with `cpufreq governors` [2] to emulate an AMP. Our test machine has Dual Quad Core Intel Xeons [1] with 8 GB RAM and runs RHEL 5.1 Desktop (Linux Kernel 2.6.18). Table 3 shows the machine configurations in detail. Our experiments with real machines use an unmodified RHEL 5.1 Desktop kernel as baseline. All experiments with the real machine use 8 threads.

Table 3: Three machine configurations

all-fast	All 8 cores are running 1.87GHz
all-slow	All 8 cores are running 1.60GHz
half-half	4 cores are running 1.87GHz, 4 cores are running 1.60GHz

6.3 Benchmarks

Our experimental benchmarks include Parsec [8] with full execution of simmedium input set. To test the robustness of scheduling policies, we select asymmetric workloads from various benchmarks suites for evaluation. Our asymmetric benchmarks (benchmarks that do not belong to the Parsec suite) are from Splash-II [23], SuperLU [11] - an Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination, and OmpSCR [12]. The detailed characteristics of the evaluated benchmarks are described in Table 4. For experiments with Age(Prof), for the Parsec benchmarks, we use simsmall input for profiling. Since smaller input sets are not available for other benchmarks, we use the same inputs for both execution and profiling. For Splash-II and OmpSCR, we use the base or standard inputs provided along with the benchmarks. For SuperLU, we use cg20.cua, g4.rua and g5.rua input files.

[6]The simulated overhead includes slowdown caused by initial cache misses that occur when a thread is context switched or migrated. Migration is assumed to cause more cache misses than context switch

7. RESULTS

7.1 Problem Demonstration in the Simulator

To verify how accurately our simulator models the real system's behavior, we compare our simulation results with Figure 1. Figure 7 shows the performance of the Parsec benchmarks on three different machine configurations using the default Linux policy and the performance on half-half configuration using the Age based policies. We use a 4 fast-4 slow configuration and a 2:1 frequency ratio for the half-half configuration to demonstrate the problem. The all-fast configuration has 8 cores with frequency 2x and the all-slow configuration has 8 cores with frequency 1x. Even though the absolute execution time for each benchmark is different from the real machine, the trends in the simulation results show that the simulator models the real system accurately. For example, the ratio of performance of blackscholes on all-fast, half-half and all-slow configurations on the real machine is 1:1.92:1.96. For the simulator, this ratio is 1:1.95:1.99, which is quite accurate. Other slow-limited benchmarks such as facesim and fluidanimate show similar performance on half-half and all-slow configurations on both the real machine and the simulator. Since the current Linux policy is asymmetry unaware, it could assign threads to the fast and slow cores in any order. To report results for the default Linux policy we run two simulations - one with threads being assigned to the fast cores first and another in which threads are assigned to the slow core first - and average the results for the two simulations. There is a minor discrepancy between the two graphs for benchmarks such as canneal and fluidanimate because of the instability of the benchmarks. For the rest of the benchmarks, the performance on half-half is between all-slow and all-fast on both real machine and simulator. Based on these results, we can safely conclude that our simulator can model asymmetric processor behavior accurately. The simulator results also show that the performance of the Parsec benchmarks on half-half configuration using Age based policies is much similar to the performance of the Parsec benchmarks on the

Table 4: Benchmark characteristics

Application	Suite	Description	Inst. Count	Locks	Barriers	# of Threads	# of Thread Groups
blackscholes	Parsec [8]	Financial Analysis	152352819	39	8	9	2
bodytrack	Parsec	Computer vision	670415058	40465	1192	9	2
canneal	Parsec	Engineering	292696036	50	0	9	2
dedup	Parsec	Enterprise Storage	1065257131	26640	0	25	4
facesim	Parsec	Animation	12261654384	14580	0	8	2
ferret	Parsec	Similarity Search	799250302	7367532	0	35	5
fluidanimate	Parsec	Animation	570968537	4168875	31998	9	2
freqmine	Parsec	Data Mining	3435161189	3083	0	8	2
streamcluster	Parsec	Data Mining	837572613	311	103984	17	3
swaptions	Parsec	Financial Analyis	503683305	39	0	9	2
vips	Parsec	Media Processing	1886383414	14876	0	11	3
x264	Parsec	Media Processing	482957580	3973	0	64	64
c_fft	OmpSCR [12]	Parallel FFT	1809116600	17	0	8	3
c_fft6	OmpSCR	Parallel FFT	2963158985	17	0	8	3
c_qsort	OmpSCR	Parallel Quicksort	1040129080	17	0	8	4
cg20.cua	SuperLU [11]	Sparse Direct Solver	2933192658	25568	0	9	7
g4.rua	SuperLU	Sparse Direct Solver	992345600	5559	0	9	9
g5.rua	SuperLU	Sparse Direct Solver	1517742541	28830	0	9	8
barnes	Splash-2 [23]	N-body method (3-D)	8309648058	1098755	136	8	5
cholesky	Splash-2	Cholesky factorization	12888835049	22049	32	8	7
fmm	Splash-2	N-body method (2-D)	9758572445	637614	272	8	6
lu_contiguous	Splash-2	LU decomposition	3117508	25	88	8	8
lu_non_contiguous	Splash-2	LU decomposition	2263332	25	88	8	7
radiosity	Splash-2	Iterative diffusion	4362156729	1657844	128	8	5
raytrace	Splash-2	3D sense rendering	100535964	74380	8	8	6

all-fast configuration than on the half-half configuration. On average, AGE and AGE(PROF) are only 15% and 9% slower than all-fast while the default Linux policy is about 59% slower.

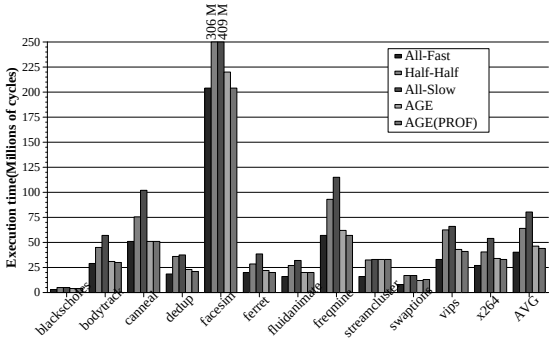


Figure 7: Behavior of default Linux scheduler and Age based policies on three machine configurations in Table 3

7.2 Evaluation

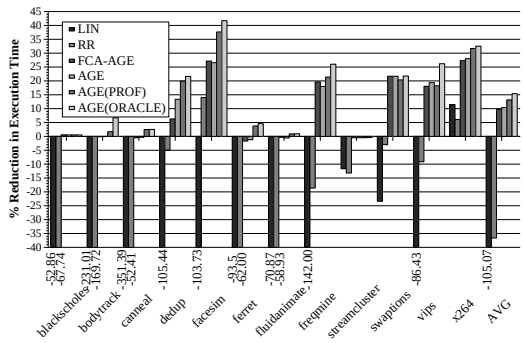
In this section, we present the results of our simulations. We compare Age based scheduling with the state-of-the-art asymmetry aware thread scheduling algorithm, SCALELD (Li's mechanism). We also compare results with the current Linux policy (LIN policy). The results we present for the Linux policy are an average of faster core first and slower core first simulations as explained in section 7.1. However, since the Linux policy performs considerably worse than SCALELD, we often omit its results. Most results are presented as percentage reduction in execution time in comparison to SCALELD. The base AMP machine configuration used for all experiments is 1 fast core and 7 slow cores with the ratio of the frequencies (performance) of the fast and slow cores being 8:1 unless otherwise specified.

7.2.1 LJFCF Policies vs. Others

Figure 8 compares the performance of different LJFCF policies with other scheduling polices. The other evaluated policies are LIN, RR and FCA-AGE (Fast Core First assignment with Age based load balancing). The results are presented as percentage reduction in execution time compared with SCALELD. Age based policies generally take less than half of the execution time of LIN due to LIN being asymmetry unaware. The RR policy also performs badly in comparison with SCALELD and Age based policies. For Parsec, compared with SCALELD, AGE shows 10% reduction in execution time, while AGE(PROF) and AGE(ORACLE) show 13% and 15% reduction in execution time, respectively. For almost all the Parsec applications, Age based mechanisms either improve the performance or give about the same performance in comparison with SCALELD. Benchmarks such as streamcluster and bodytrack which have frequent barriers (short distances between milestones) provide little scope for improvement. Canneal has a very small parallel section, hence does not show much benefit. Blackscholes does not provide any benefit since the default timeslice value assigned to each thread is very large for reassignment to happen. Reassignment gives each thread an opportunity to execute on the fast cores. By reducing the time slice value we can obtain improvement for blackscholes. Note that all experiments used the same time slice value as mentioned in section 6. The right of Figure 8 shows the reduction in execution time with different polices for asymmetric benchmarks. For asymmetric benchmarks, AGE, AGE(PROF) and AGE(ORACLE) provide an improvement of 8%, 9% and 13% respectively, with c_fft, c_fft6, cg20.cua (SuperLU) and cholesky showing significant improvement. Though Age based scheduling assumes that the threads of an application are symmetric, even asymmetric benchmarks show significant improvement with the Age based policies. This is because Age based scheduling gives opportunities to different threads to execute on fast cores by migrating threads to and from fast cores and does not keep the same threads assigned to fast cores as SCALELD. In general, FCA-AGE performs similar to AGE because the Fast Core First assignment policy of FCA-AGE is used only when new threads are created, thereafter, AGE is used for reassignment of threads on events such as core going idle, reassignment timer expiry and so on. These trigger events for reassignment happen frequently enough that the effects of poor initial assignment by FCA are not very visible.

7.2.2 Idle Cycles

Figure 9 shows the idle cycles of each core for different policies for each Parsec benchmark. By analyzing this idle cycle distribution we can conclude how much each policy utilizes fast cores.



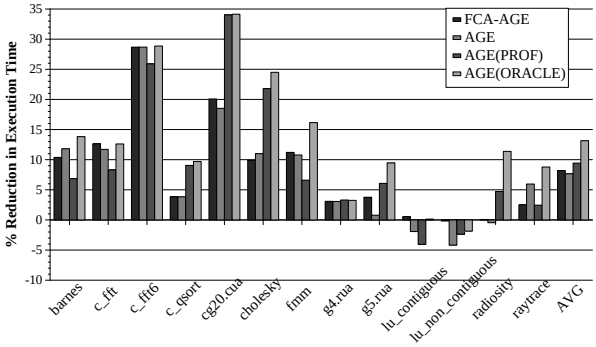


Figure 8: Age based policy with other scheduling policies (Left: Parsec benchmarks, Right: Asymmetric benchmarks)

The Age based policies try to utilize fast cores as much as possible. Since the Linux scheduler distributes threads equally across cores, it results in under utilization of fast cores (Core 0 is the fast core in this configuration). Both SCALELD and the Age based policies keep the fast core busy at all times. While SCALELD keeps the same threads assigned to the fast cores, the Age based policies try to give all threads opportunities to execute on the fast cores. Figure 10 shows the reduction in total idle cycles of all cores for the Parsec benchmarks for AGE, AGE(PROF) and AGE(ORACLE) policies in comparison with SCALELD. On average, AGE, AGE(PROF) and AGE(ORACLE) result in 29%, 40% and 48% reduction in total idle cycles. This shows that the Age based policies have better core resource utilization.

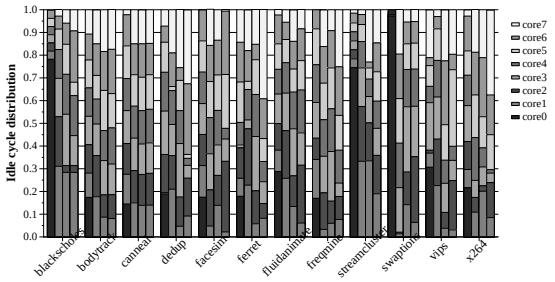


Figure 9: Normalized distribution of idle cycles for each core - The column order from left is LIN, SCALELD, AGE and AGE(ORACLE) policies

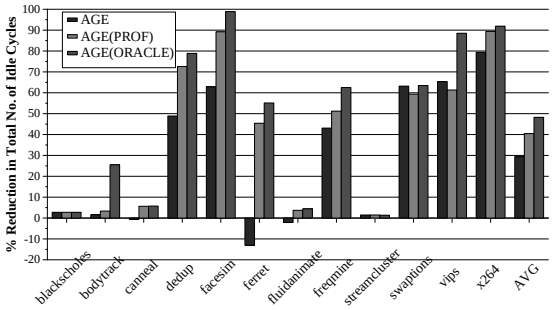


Figure 10: Total idle cycles of all cores

7.2.3 *Comparison between Age Based Scheduling and Other Policies*

Table 5 shows the comparison between the current scheduling policy in Linux, Li's mechanism and the Age based policies on AMPs. While the Linux scheduling policy performs badly on all counts, SCALELD performs relatively well. But, SCALELD allows threads that have been assigned to fast cores remain there unlike the Age based policies, which give different threads opportunities to execute on the fast cores and thus exploit fast cores better.

7.2.4 *Prediction Accuracy*

Figure 11 and Figure 12 show the prediction accuracy of the Age based policies for the Parsec and asymmetric benchmarks. The accuracy is calculated as ratio of the number of correct assignments made to the number of assignments made. An assignment of a thread to a core is correct if the thread is assigned to the same kind of core (fast or slow) as it would have been by AGE(ORACLE) i.e., oracle based policy.[7] Age based policies perform assignment of threads when (1) a thread is created, (2) a core goes idle, or (3) when the periodic reassignment timer expires. For the Parsec benchmarks, AGE has an accuracy of 73%. Expectedly, AGE(PROF) has better accuracy than AGE since it makes predictions based on profile information; its accuracy for the Parsec benchmarks being 88%. AGE and AGE(PROF) have accuracies of 72% and 84% respectively for asymmetric benchmarks.

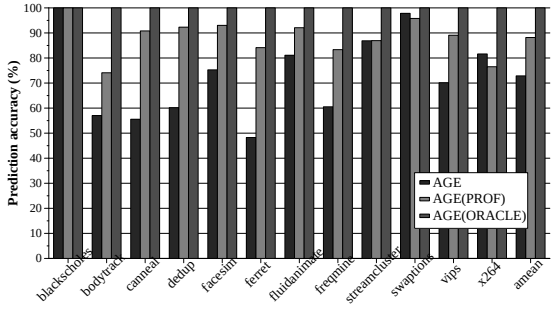


Figure 11: Prediction accuracy of Age based policies for the Parsec benchmarks

7.2.5 *Longest Job First vs. Shortest Job First*

We evaluate whether Longest Job First (LJF) to fast core or Shortest Job First (SJF) to fast core is suitable for Age based scheduling on AMPs. SJF [22] is typically used in batch systems to increase system throughput. In our context, a job represents the remaining execution distance until the next milestone. To eliminate the effect

[7]For accuracy results, we allow a tolerance of -5% to +5%. For example, instead of thread A, if another thread B whose remaining execution time is at least 95% or at most 105% of A is assigned, we consider it to be an accurate assignment

Table 5: Comparison between Policies on AMPs

Scheduling policy	Fairness	Utilization of fast cores	Exploitation of fast cores	Application stability
LIN	No	No	No	No
SCALELD	Yes	Yes	No	Yes
Age based polices	Yes	Yes	Yes	Yes

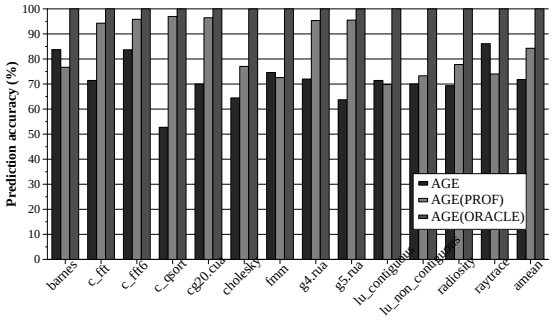


Figure 12: Prediction accuracy of Age based policies for Asymmetric benchmarks

of predicting task/thread distances inaccurately, we use the oracle to find threads with the longest and the shortest distances in this section. Figure 13 shows the simulation results for both Age based Longest Job Fast Core First (LJFCF or LFJ based Age scheduling) and Shortest Job Fast Core First (SJFCF or SJF based Age scheduling) policies. For each policy, we run experiments with both Idle Core wake up (IW) and Suitable Core wake up (SW) assignment policies. Though for certain benchmarks such as blackscholes and streamcluster there is negligible performance difference between LJFCF and SJFCF, on the whole, LJFCF performs considerably better than SJFCF. For the Parsec benchmarks, LJFCF performs about 15% better than the baseline, where as SJFCF is only about 3% better than the baseline. Similar behavior is shown by LJFCF and SJFCF for asymmetric benchmarks also. Hence, we conclude that on AMPs, longest job fast core first is a better policy for multi-threaded applications. Figure 13 also shows that on average, there is not much difference between using IW and SW wake up assignment policies.

7.2.6 Milestones Used by LJFCF for Predictions

In LJFCF, we can predict the remaining execution time (or distance) either to the next BT (barrier or termination) or to the next BCJT (barrier, critical section, join and termination). Figure 14 shows the results for LJFCF using different combinations of BT and BCJT for thread assignment and reassignment. The oracle provides the distance to the next milestone for both BT and BCJT. The results show that using BT is slightly better than BCJT (15% performance benefit vs. 12% for the Parsec benchmarks). The main reason is that in BCJT, for benchmarks like dedup that have many locks, most threads are treated as having short distances to milestones due to locks. Since most (more than 90% for the Parsec benchmarks) locks can be acquired without contention i.e., without the thread getting blocked, treating a thread that has many locks as having short distances could result in the thread not getting sufficient opportunities to execute on the fast cores, resulting in application slowdown compared to BT. On top of that, predicting BT is simpler than BCJT. Hence, we decide to use BT as our remaining execution time calculation method.[8] The policies used for this set of experiments are shown in Table 6.

[8]Except for the experiments in this section, our experiments with LJFCF always use BT

Table 6: Scheduling mechanisms

Name	Combination
AGE(ORACLE)-B	Age(Oracle) with BT for assignment and BCJT for reassignment
B-AGE(ORACLE)	Age(Oracle) with BCJT for assignment and BT for reassignment
B-AGE(ORACLE)-B	Age(Oracle) with BCJT for both assignment and reassignment

7.3 Microarchitecture Sensitivity Study

To know how sensitive our scheduling policy is to the heterogeneity of asymmetric configurations, we vary different parameters of the asymmetric configurations.

7.3.1 Different Number of Fast and Slow Cores

In Figure 15, we vary the number of fast and slow cores. While the relative performance of the Linux scheduler (not shown in Figure 15) degrades considerably as we reduce the number of fast cores from 4 to 1, the Age based scheduling policies show better relative performance as the number of fast cores is reduced. Though Age based policies are better than SCALELD at exploiting fast cores, as the number of number of fast cores is increased more threads get to take advantage of fast cores in SCALELD, hence the performance difference between Age based policies and SCALELD reduces as the number of fast cores is increased. Remember that in SCALELD once a thread is assigned to a core, it continues to execute on the same core. We believe AMPs will have very few number of fast cores and several slow cores, and for such configurations the Age based policies are a better alternative than SCALELD. Note that the performance of Age based policies for each configuration is normalized to the performance of SCALELD for the same configuration.

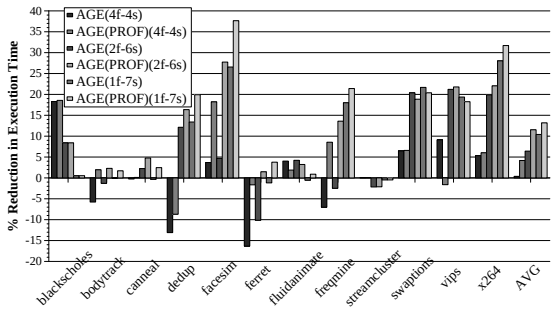


Figure 15: Different combinations of AMPs

7.3.2 Different Frequency Ratio between Fast and Slow Cores

Figure 16 shows the result of varying the frequency ratio between fast cores and slow cores. In Figure 16, PAR and ASY represent the average of the normalized execution times for the Parsec and the asymmetric benchmarks, respectively. As we make the cores more asymmetric, performance of LIN degrades drastically when compared with SCALELD or the Age based policies. When the frequency ratio is 2:1, the Age based policies perform similar to SCALELD but as we make cores more asymmetric, the benefit of Age based policies increases. The results show that as we have higher asymmetric characteristics, the need for asymmetry aware

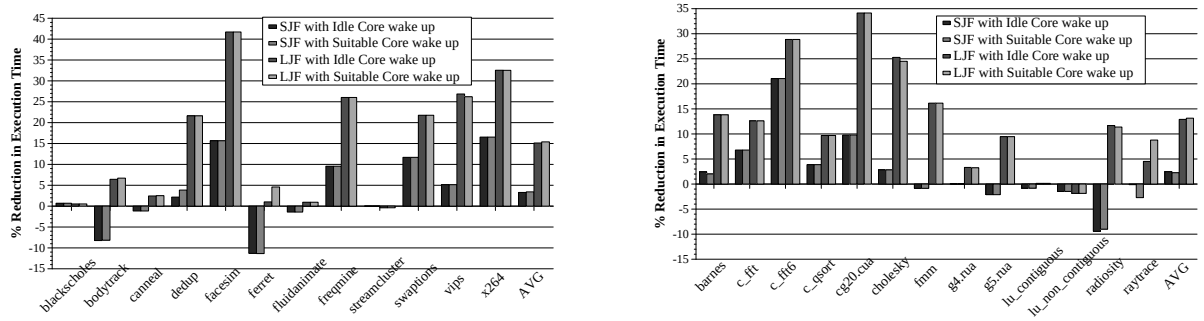


Figure 13: Performance comparison between LJFCF and SJFCF (Left: Parsec benchmarks Right: Asymmetric benchmarks)

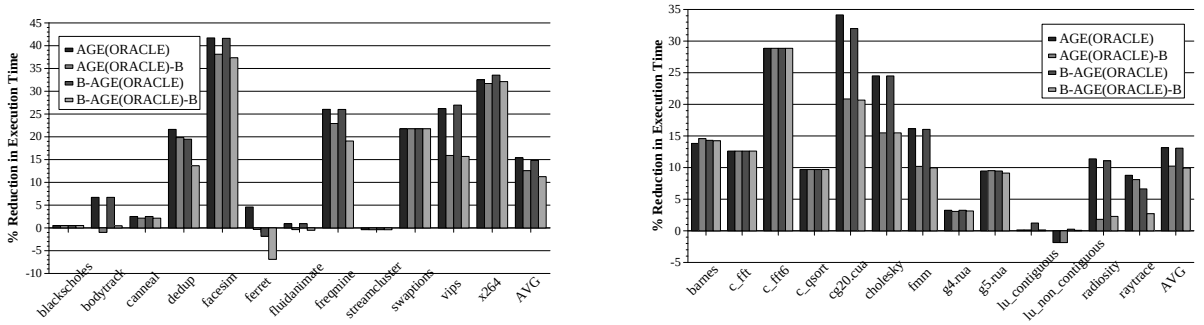


Figure 14: Remaining execution time calculation method BT or BCJT (Left: Parsec benchmarks, Right: Asymmetric benchmarks)

thread scheduling increases and also that Age based scheduling performs better than SCALELD.

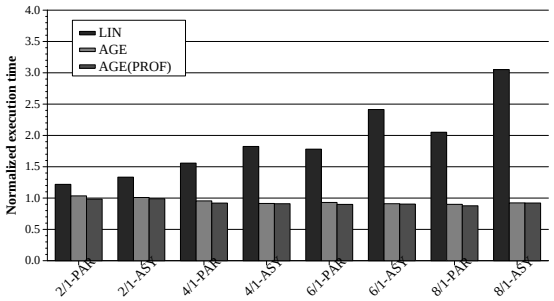


Figure 16: Different frequency configurations for AMPs

8. RELATED WORK

Many researchers have looked at several different aspects of thread scheduling on AMPs. Balakrishnan et al. [6] evaluated the performance of multithreaded applications on SMPs and AMPs and showed that application performance becomes unstable and less scalable on AMPs. They also showed that an operating system (or an user level scheduler) that is aware of the asymmetry in core performance can eliminate unpredictability in application performance. Another important result of their work was that AMPs can give better performance than a SMP with all slow cores. Though they implemented an asymmetry aware scheduling algorithms for AMPs for their experiments, their main idea was not thread scheduling, hence they did not focus on designing strong scheduling algorithms.

Grant and Asfahi [14] studied the power-performance efficiency of AMPs using a new scheduling algorithm for AMPs and concluded that an asymmetry aware scheduling algorithm can provide considerable savings in energy while maintaining performance comparable to a SMP with all fast cores. Their focus was on designing a power efficient algorithm for Hyper-Threaded AMPs while providing good application performance. Our goal is to design a scheduling algorithm for AMPs that provides high performance.

Cai et al. [10] described a mechanism called meeting points that dynamically detects critical threads in a parallel region and tries to boost the application performance by giving priority to the critical thread in an SMT. Their mechanism is not suitable for general thread scheduling and works only for parallel regions. Annavaram et al. [5] applied EPI throttling techniques to show that for certain applications an AMP can provide better performance than an SMP that consumes the same amount of power as the AMP. Their goal was to maximize the performance of an AMP for a given power budget.

Fedorova et al. [13] proposed a self-tuning algorithm based on reinforcement learning for thread scheduling. Our work does not use any machine learning techniques and their work also does not include any experimental results for comparison.

Scheduling using Architectural Signatures [21] assigns threads to cores based on the information contained in the architectural signatures, This kind of scheduling results in only few threads getting the opportunity to execute on fast cores, unlike in Age based scheduling where all threads get opportunities to execute on fast cores.

Becchi and Crowley [7] propose an IPC driven dynamic assignment of threads for Heterogeneous Multiprocessors. Their mechanism switches threads between fast and slow cores and monitors the IPC of threads on both kinds of cores to decide which threads should be assigned to fast cores. Kumar et al. [16] propose a mechanism which does sampling of application behavior on different cores before deciding on the assignment of threads to cores. These

mechanisms require monitoring of execution of applications on different cores with lot of switching or migrations and they are also incomplete in that they does not deal with issues such as load balancing.

The closest to our work is the work by Li et al. [17]. They proposed asymmetry aware thread assignment and load balancing policies that follow the faster core first principle and try to keep the load on each core proportional to its compute power. Their policy tries to ensure that the fast cores are not underutilized, but does not take any measures to ensure that critical threads or threads that are lagging behind get an opportunity to make faster progress or catch up.

Our policy recognizes that in an AMP threads will not make uniform progress and some threads will lag behind others. We identify critical threads or threads that are lagging behind and assign them to fast cores so that they can catch up and all threads can complete at the same time.

9. CONCLUSION AND FUTURE WORK

We have proposed and evaluated a new asymmetry aware thread scheduling policy for AMPs. The two variations (prediction and profiling) of our scheduling policy, Age Based Longest Job Fast Core First, more than double the performance compared to today's Linux scheduler. The prediction based mechanism provides an improvement of 10.4% on average for the Parsec benchmarks and 7.6% for the asymmetric benchmarks over the state-of-the-art asymmetry aware thread scheduling policy. On the other hand, the profiling based mechanism provides improvements of 13.2% and 9.4%. Using the proposed policy, we are able to obtain a benefit of up to 37% over the state-of-the-art asymmetry aware scheduling policy. We evaluate our scheduling policy using various workloads and various machine configurations with a wide range of applications. We also thoroughly characterize the workload to provide an insight on how much the workloads are symmetric and asymmetric.

Future work can be the development of a run-time feedback system to monitor the progress of threads. The prediction mechanism can be further improved to improve the performance benefits.

Acknowledgments

We thank the anonymous reviewers for their comments. We also thank Tong Li, Aater Suleman and Aniruddha Dasgupta for their feedback on improving the paper. We thank Richard Vuduc for providing us with the SuperLU benchmark. We gratefully acknowledge the support of Intel Corporation and Microsoft Research.

10. REFERENCES

[1] Intel xeon processor. http://www.intel.com/support/processors/xeon/.

[2] Linux kernel CPUfreq subsystem. http://www.kernel.org/pub/linux/utils/kernel/cpufreq/cpufreq.html.

[3] O(1) Scheduler. http://joshaas.net/linux/.

[4] Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor–White Paper, March 2004.

[5] M. Annavaram, E. Grochowski, and J. Shen. Mitigating Amdahl's Law through EPI Throttling. In *ISCA-32*, 2005.

[6] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. The Impact of Performance Asymmetry in Emerging Multicore Architectures. In *ISCA-32*, 2005.

[7] M. Becchi and P. Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *Proceedings of the 3rd conference on Computing Frontiers*, 2006.

[8] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. Technical Report TR-811-08, Princeton University, 2008.

[9] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel, Third Edition*. O'Reilly, 2005.

[10] Q. Cai, J. Gonzalez, R. Rakvic, G. Magklis, P. Chaparro, and A. Gonzalez. Meeting points: Using thread criticality to adapt multicore hardware to parallel regions. In *PACT '08*, New York, NY, USA, 2008. ACM.

[11] J. W. Demmel, J. R. Gilbert, and X. S. Li. An Asynchronous Parallel Supernodal Algorithm for Sparse Gaussian Elimination. *SIAM J. Matrix Analysis and Applications*, 20(4):915–952, 1999.

[12] A. J. Dorta, C. Rodriguez, F. D. Sande, and A. Gonzalez-Ecsribano. The OpenMP Source Code Repository: an Infrastructure to Contribute to the Development of OpenMP.

[13] A. Fedorova, D. Vengerov, and D. Doucette. Operating System Scheduling On Heterogeneous Core Systems. Technical report, Sun Microsystem, 2007.

[14] R. Grant and A. Afsahi. Power-Performance Efficiency of Asymmetric Multiprocessors for Multi-threaded Scientific Applications. In *IPDPS*, 2006.

[15] Intel Corporation. *Intel VTune Performance Analyzers*. `http://www.intel.com/vtune/`.

[16] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *ISCA-31*, 2004.

[17] T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. Efficient Operating System Scheduling for Performance-Asymmetric Multi-Core Architecture. In *Proceedings of Supercomputing 07*, 2007.

[18] R. Love. *Linux Kernel Development, Second Edition*. Novell Press, 2005.

[19] M. K. McKusick and G. V. Neville-Neil. Thread Scheduling in FreeBSD 5.2. *Queue*, 2(7):58–64, 2004.

[20] T. Y. Morad, U. C. Weiser, A. Kolodny, M. Valero, and E. AyguadÃl'. Performance, Power Efficiency and Scalability of Asymmetric Cluster Chip Multiprocessors. *Computer Architecture Letters*, 5(1), 2006.

[21] D. Shelepov and A. Fedorova. Scheduling on Heterogeneous Multicore Processors Using Architectural Signatures. In *WIOSCA*, 2008.

[22] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems Design and Implementation, Third Edition*. Prentice Hall, 2006.

[23] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA-22*.