No More Backstabbing... A Faithful Scheduling Policy for Multithreaded Programs

Kishore Kumar Pusukuri, Rajiv Gupta, Laxmi N. Bhuyan
Department of Computer Science and Engineering
University of California, Riverside
Riverside, USA 92521
kishore@cs.ucr.edu, gupta@cs.ucr.edu, bhuyan@cs.ucr.edu

***Abstract*—Efficient contention management is the key to achieving scalable performance for multithreaded applications running on multicore systems. However, contention management policies provided by modern operating systems increase context-switches and lead to performance degradation for multithreaded applications under high loads. Moreover, this problem is exacerbated by the interaction between contention management policies and OS scheduling polices. Time Share (TS) is the default scheduling policy in a modern OS such as OpenSolaris and with TS policy, priorities of threads change very frequently for balancing load and providing fairness in scheduling. Due to the frequent ping-ponging of priorities, threads of an application are often preempted by the threads of the same application. This increases the frequency of involuntary context-switches as wells as lock-holder thread preemptions and leads to poor performance. This problem becomes very serious under high loads.**

To alleviate this problem, in this paper, we present a scheduling policy called Faithful Scheduling (FF), which dramatically reduces context-switches as well as lock-holder thread preemptions. We implemented FF on a 24-core Dell PowerEdge R905 server running OpenSolaris.2009.06 and evaluated it using 22 programs including the TATP database application, SPECjbb2005, programs from PARSEC, SPEC OMP, and some microbenchmarks. The experimental results show that FF policy achieves high performance for both lightly and heavily loaded systems. Moreover it does not require any changes to the application source code or the OS kernel.

***Keywords*-Scheduling; priorities; contention; context-switches**

I. Introduction

The advent of multicore architectures provides an attractive opportunity for achieving high performance for a wide variety of multithreaded applications. However, exploiting the system density, and the parallelism they offer, to improve performance of multithreaded applications is a challenging task. This is because multithreaded application performance is sensitive to the implementations of synchronization primitives and contention management policies. Therefore the key to achieving high performance for multithreaded applications running on multicore systems is to use appropriate synchronization primitives along with efficient contention management policies. Contention management policies are either based on spinning, or blocking, or a combination of both. Spinning resolves contention by busy waiting, therefore waiting threads respond to lock handoffs very quickly. However, spinning threads can wastes CPU resources and prevent the lock-holder thread from running and releasing the lock [1], [3], [6]. This dramatically degrades performance and becomes a prominent problem in systems under high load conditions. In contrast, the blocking scheme reschedules waiting threads and allows other threads to use the system resources. However, blocking scheme increases context-switches, overloads OS scheduler, and thus leads to poor performance [1], [3], [6].

To alleviate the above problems with spinning and blocking, several hybrid schemes have been introduced. The adaptive mutex provided by OpenSolaris [3], Linux futex [17], and pthread mutex provided by pthread library are examples of such hybrid schemes. Both Solaris adaptive mutex and Linux futex use the state-of-the-art spin-then-block contention management policy. According to this policy, threads spin if the lock-holder thread is running on another CPU and block otherwise. This policy is based on the assumption that mutex hold times are typically short enough that the time spent spinning is less than the time it takes to block [3]. However, this policy faces challenges in providing optimal balance between spinning and blocking because this balance must change with increasing core and thread counts [1], [2].

Next we illustrate the above problem using the SPEC OMP program applu. Fig. 1 shows the speedup and the

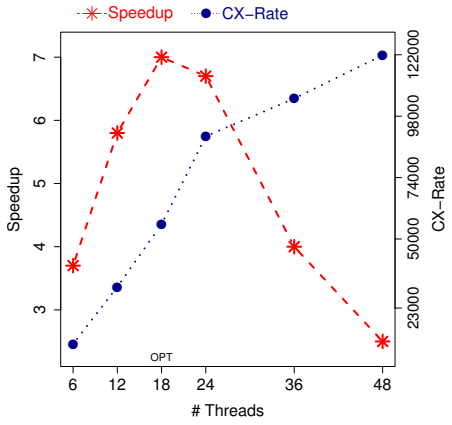


Figure 1: Speedup of applu degrades while CX-Rate increases as thread count grows on a 24-core machine. 24 threads represents 100% load.

context-switch (CX) rate observed by running applu on 24-core machine for varying number of threads. The speedup is computed relative to the serial execution-time. Applu achieves the best performance with 18 threads on our 24-core machine. As we can see, speedup of 'applu' drops while CX-Rate increases as thread count grows. The implementation of applu is based upon pthreads and pthread mutex uses the state-of-the-art spin-then-bock contention management. As we discussed above, as load (thread count) increases, the spin-then-block policy increases the CX-Rate, overwhelms the OS scheduler, causing poor performance even with 75% load (.e., 18 threads). Applu is a contention-bound program that experiences high CX-Rate and spends around 47% of its elapsed time in lock-contention even with #threads $<$ #cores, i.e., less than 100% load.

The CX-Rate increases further because of the unwanted interactions between the spin-then-block policy and the Time Share (TS) scheduling policy which is the default scheduling policy in a modern OS. With TS scheduling policy, priorities of threads change very frequently for balancing load and providing fairness in scheduling. Priority adjustments are made based on the time a thread spends waiting for processor resources, consuming processor resources, etc. [3]. Therefore, at any execution point of a multithreaded application, some of the threads belonging to the application get higher priority while the others get lower priority. This leads to preemption of low-priority threads by the high-priority threads of the same application which often includes lock-holder thread preemptions. This is what we call "Backstabbing" (BS) which leads to increased frequency of involuntary context-switches (ICXs), i.e. context-switches that cause threads to be involuntarily taken off a core. Whenever a lock-holder thread is preempted, the threads that are spinning for that lock will be blocked, which in turn increases voluntary context-switches (VCXs), i.e. context-switches that happen when a threads fail to acquire a lock or are blocked due to IO. The changes in context-switch rates lead to further changes in thread priorities. Thus, the interaction between the state-of-the-art spin-then-block policy and the TS scheduling policy creates a vicious cycle between priority changes and context-switches, which causes a drastic increase in CX-Rate (ICX-Rate + VCX-Rate) with increasing load; thus leading to poor performance.

To alleviate the problems with the state-of-the-art contention management policies, Johnson et al., [1] proposed a "load control" mechanism that decouples load management from contention management. This approach uses blocking to control the number of runnable threads and then spinning in response to contention. Although this approach works well, it needs to modify the applications for making spin locks visible, it is sensitive to spikes in the load, and it does not function well when priority inversions occur due to nested critical sections [1]. Moreover, the implementation of the load controller uses 7 ms as an update interval, with which, it is difficult to obtain accurate processor-usage statistics, and the overhead increases linearly with the number of threads [1].

However, unlike the above approach, in this paper we present a new scheduling policy called faithful scheduling (FF), where all threads of an application have same priority for the entire execution. FF allocates the same time-quantum to all the threads belonging to one application; however, its value varies according to application's usage of system resources. By providing same priority to all the threads of an application, this policy completely eliminates BS, breaks the vicious cycle between thread priority changes and context-switches, dramatically reduces CX-Rate, and thus leads to high performance. By completely eliminating BS, FF policy makes all the threads of an application fair to each other. FF policy is agnostic to dynamic load changes and improves performance predictability. The overhead of the FF policy is negligible and it is an attractive approach as it requires no changes to the application source code or the OS kernel. Moreover, since it completely avoids priority inversion problems and thus handles nested critical sections well.

We implemented FF on a 24-core Dell PowerEdge R905 server running OpenSolaris and evaluated it using 22 programs including the TATP database application [24], SPECjbb2005 [27], programs from PARSEC [26], SPEC OMP [27], and a microbenchmark [1]. The experimental results show that at 100% load, FF policy achieves more than 10% performance improvement for five programs with a maximum of 35% improvement, 4%-10% for six programs, less than 4% for nine programs, and there is no improvement for one program over TS policy. At 200% load, FF policy achieves more than 10% performance improvement for eight programs with a maximum of 107% improvement, 4%-10% for six programs, less than 4% for seven programs over TS policy. Furthermore, FF policy also achieves performance improvements under light loads, i.e., less than 100% load.

The key contributions of this work are as follows:

- We identify the reasons behind the problems caused by the interactions between the spin-then-block policy and TS scheduling policy through an in-depth performance analysis of several multithreaded programs on a 24-core multicore system.
- We present a scheduling policy FF, which eliminates lock-holder thread preemptions, dramatically reduces context-switches over TS policy, and achieves high performance for a wide variety of benchmarks for both lightly and heavily loaded systems.
- Finally, we develop FF policy using simple utilities available on a modern OS and it requires no changes to the application source code or the OS kernel. It is very effective against phase changes of the application, it completely avoids spikes in the load, and improves performance predictability. Moreover, it introduces negligible runtime overhead.

The remainder of this paper is organized as follows. Section II explains the problems caused by the interactions between OS scheduling and contention management policies. Section III presents the implementation of FF policy in detail and Section IV presents the experimental setup. Section V describes the evaluation of FF policy against a wide variety of benchmark programs. Related work and conclusions are given in Sections VI and VII.

II. Interaction between OS Scheduling and Contention Management

This section explains how the interaction between contention management policies and OS scheduling policies hurts the performance of multithreaded programs running on multicore systems.

Time Share (TS) is the default scheduling policy in a modern OS such as OpenSolaris. With TS scheduling policy, priorities of threads change very frequently for balancing load and providing fairness in scheduling. Priority adjustments are made based on the times a thread spends waiting for processor resources, consuming processor resources, etc [3]. Therefore, at a given point in time, some of the threads belonging to an application get higher priority while the others get lower priority. This leads to preemptions of the low-priority threads of an application by the high-priority threads of the same application, i.e. 'Backstabbing' (BS). BS often includes lock-holder thread preemptions which increases the ICX rate. We can further divide ICX into two types: time-quantum context-switches (TQE ICX) happen because of time-quantum expiration; and preemption context-switches happen when a higher priority thread preempts a lower priority thread (HPP ICX).

As we can see in Fig. 2(a), applu program experiences a high degree of HPP ICX (56% of total ICX) when it is run with 24 threads on 24 cores (100% load). Using DTrace [5] scripts, we observed that almost all of these HPP ICX are caused by applu threads i.e., applu experiences around 55% BS at 100% load. Here BS is specifically defined as % of HPP ICX caused by the same application threads. This is because HPP ICX is also caused by high priority system processes

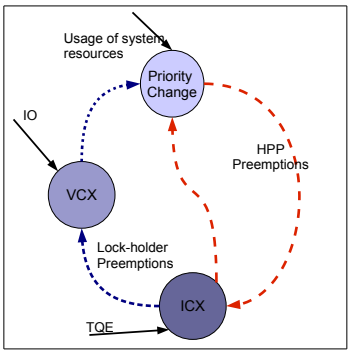
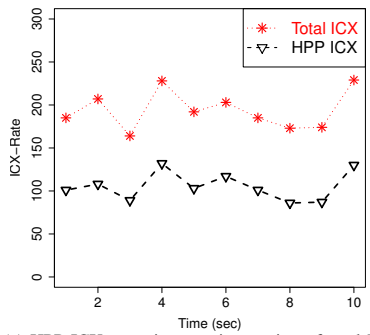


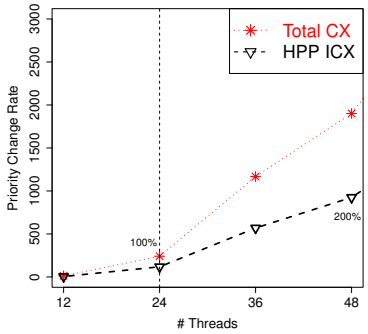
Figure 4: The interactions between the TS policy and the spin-then-block policy create vicious cycles between priority changes and context-switches.

running along with the application threads. However, we can expect that BS is the major portion of HPP ICX (i.e., HPP ICX $\sim$ BS) when load crosses 100%. As shown in Figure 2(b), priority change-rate increases as load increases and also a major portion of priority changes are due to HPP ICX. Another important point to note is that ICX (HPP ICX and TQE ICX) causes a major portion of VCX. Figure 2(c) shows a drastic increase in HPP ICX as load crosses 100%. Therefore, we can expect that the frequency of lock-holder thread preemptions will increase once load crosses 100%. Thus, frequent ping-ponging [3] of thread priorities increases HPP ICX, specifically BS, which in turn increases CX-Rate (ICX-Rate + VCX-Rate), and ultimately vicious cycle is created between context-switches and priority changes.

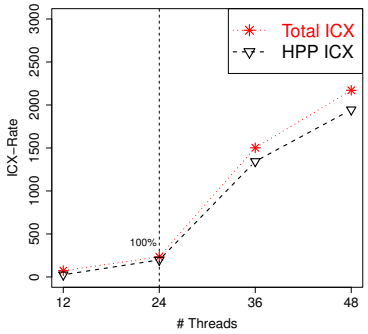
As shown in Fig. 4, the TS policy changes priorities of threads based on their usage of system resources. Frequent ping-ponging of thread priorities leads to HPP ICX, i.e., force the threads off the CPU, which often include lock-holder threads. When a lock-holder thread is preempted then all the threads that are waiting for that lock will be blocked, i.e., generates VCX. Then threads will join the lock's sleep queue and their priorities will be changed based on their waiting time in the sleep queue. Thus, this process repeats continuously, increasing CX-Rate and priority change-rate, and thus leads to poor performance.



(a) HPP ICX occupies a major portion of total ICX.



(b) HPP ICX leads to changes in thread priorities.



(c) Drastic increase in HPP ICX as load crosses 100%.

Figure 2: Frequent changes in thread priority drastically increases context-switches and in turn context-switches lead to changes in thread priority. A vicious cycle is created between priority changes and context-switches.

A. *Lock-contention vs Backstabbing (BS)*

From the above observations, we can expect that high contention applications suffer more from BS than contention-free applications. This is because threads of high contention application seriously compete for lock acquisitions leading to high CX-Rate. Contention-free applications scale well and typically they experience CX-Rate far lower than high contention applications. To get a clear idea about this, we ran three different benchmark programs (nearly contention-free, medium contention, and high-contention) and observed how BS varies along with thread count. Fig. 3 shows the results.

As shown in Fig. 3(a), swaptions is a nearly contention-free program and it does not significantly suffer from BS. BS is almost nil when the load is below 100% and small under high loads. This is because when the load crosses 100%, there are more chances of lock-holder thread preemptions and also high HPP ICX. However, this becomes a prominent problem for the high contention programs. As shown in Fig. 3 (b) and (c), programs fluidanimate and applu experience high % of BS. As applu is a high contention program, it suffers from high % of BS even under low loads. These observations demonstrate two things: (1) BS rapidly increases under high loads, specifically when the load crosses 100%, and (2) high contention programs experience significant BS even when the load is below 100%. Therefore, if we completely avoid BS then we can minimize CX-Rate and improve performance. In order to avoid BS completely, we need to break the vicious cycle between priority changes and context-switches.

Thus, based on the above observations, in the next section, we present a scheduling policy called faithful scheduling (FF), which breaks the cycle between priority changes and context-switches, completely eliminates BS, dramatically reduces CX-Rate, and thus leads to higher performance.

III. Faithful Scheduling Policy (FF)

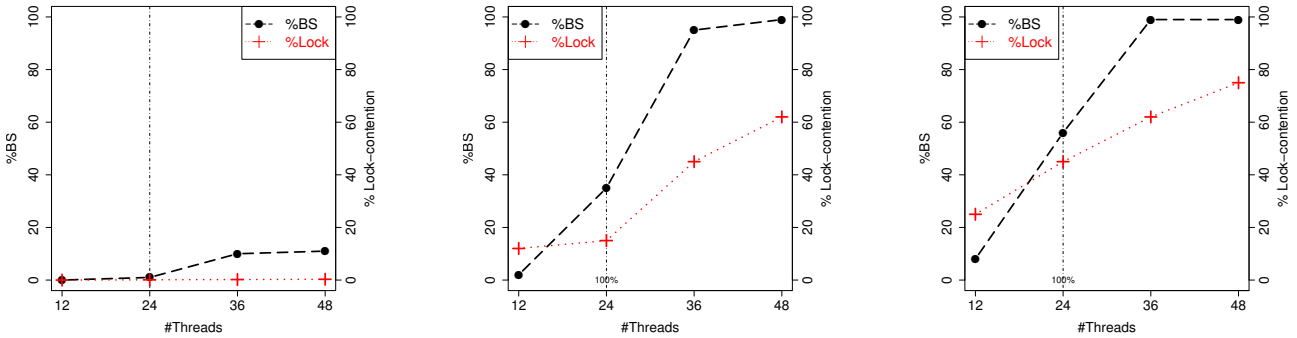
The previous section highlights the fact that the interactions between contention management and OS scheduling create vicious cycle between priority changes and context-switches, which leads to poor performance. Therefore, to break the vicious cycle and achieve high performance, we propose a scheduling policy called Faithful Scheduling Policy (FF) with the following key characteristics:

1) Same priority is assigned to all the threads of a given application.
2) Time-quantum is allocated based on the resource usage of the entire application, specifically based on lock-contention and cache miss-ratio of the application.

By providing same priority to all the threads of an application, FF policy completely avoids BS, dramatically reduces CX-Rate, and leads to high performance. Since priorities of all the threads of an application are same, FF allocates equal time-quantum to all of them for reducing unwanted TQE ICX. Moreover, this makes all the threads of an application fair to each other. However, finding the right time-quantum for an application is tricky. For this, via extensive experimentation with a wide variety of benchmarks, we derived a metric called "scaling-factor" and developed a scaling-factor table that guides time quantum allocation.

A. *Scaling-factor Table*

Finding right time-quantum is very important to provide fair allocation of CPU cycles for all the threads of a multi-threaded application. Threads of a CPU-intensive and low contention application heavily compete for CPU resources. Therefore, it is appropriate to provide small time quantum for both CPU-intensive and low contention application threads. In this way no thread will wait for a long time for a CPU. In contrast, it is appropriate to provide large time-quantum for both high-contention and memory-intensive application threads. In case of high-contention applications, large time-quantum allows lock-holder thread to complete its work quickly, release the lock, and allow other threads to make progress. Moreover large time-quantum for contention bound application threads reduces unwanted TQE ICX and also reduces the lock acquisition overhead since a wakeup and a context-switch are required before the blocking thread can become the owner of the lock it requires [3]. Based on the above observations, the metric scaling-factor is defined in Eq. (1).



(a) swaptions (No contention). (b) fluidanimate (Medium contention). (c) applu (High contention).

Figure 3: Lock-contention vs BS (24 threads is 100% load).

Scaling-factor = 1 - max(Miss-ratio, Lock-contention) (1)

Where 'Miss-ratio' is last-level cache miss-ratio and 'Lock-contention' is the percentage of time application threads spend waiting for user locks, condition-variables, etc. Using Miss-ratio we can identify whether an application is memory-intensive or not.

By conducting experiments with a wide variety of multi-threaded programs and different time-quanta, we developed the scaling-factor table shown in Table I, in which the time-quantum goes down as the scaling-factor goes up (inspiration from the priority dispatcher tables [3] of modern OS). More specifically, to derive the table, first we categorize the applications as memory intensive, CPU intensive, high contention, or low contention applications. Then we selected a few of applications from each category -- a total of 8 out of 22 applications, and ran them with varying time-quantum ranging from 10 ms to 400 ms. The scaling factor table obtained was then used in our experiments for all 22 applications. The 8 applications used to populate the scaling factor table are: streamcluster, swim, swaptions, ferret, apsi, applu, art, and bodytrack.

Therefore, based on the application's cache miss-ratio and lock-contention, scaling-factor of the application is between one and zero. For scalable applications such as CPU-intensive and low-contention applications, scaling-factor is high and close to one, and for non-scalable applications such as high memory-intensive or high lock-contention applications, scaling-factor is close to zero. One important point here is that the scaling-factor value is for the entire application not per thread. Based on the scaling-factor value, FF policy allocates corresponding time-quantum to all the threads of the application.

Table I: The Scaling-factor Table. The range of the scaling-factor is 0.10.

scaling-factor	TQ(ms)
(0.01 -- 0.10)	250
(0.11 -- 0.20)	200
(0.21 -- 0.30)	150
(0.31 -- 0.40)	120
(0.41 -- 0.50)	100
(0.51 -- 0.60)	80
(0.61 -- 0.70)	50
(0.71 -- 0.80)	30
(0.81 -- 0.90)	20
(0.91 -- 1.00)	10

B. Dealing with phase changes

Some applications have multiple different phases or regions and exhibit different usages of system resources over time. Therefore, we need to continuously monitor the applications and apply appropriate time-quantum according to the resource usage of their current phase. However, among the 22 benchmark programs we studied, only a couple of programs ammp and SPECjbb2005 show significantly two different phases in their execution. For example, consider ammp

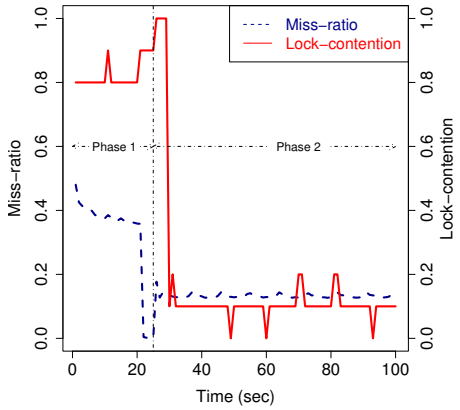


Figure 5: Phase changes of ammp. Here ammp is run with 24 threads. Lock-contention value 1 means application experiences lock-contention for 100% of the total elapsed time.

SPEC OMP program. As shown in Fig. 5, ammp has two significantly different phases. While ammp experiences high miss-ratio and high lock-contention in Phase-1 (i.e., for the first 25 seconds), it experiences low miss-ratio and low lock-contention in Phase-2. Therefore according to the scaling-factor table, FF policy allocates large time-quantum for the first 25 seconds, and small time-quantum for the rest of its execution.

C. Dealing with pipeline parallelism

It is fine to allocate equal time-quantum to all the threads of an application based on pure data-parallelism. This is because, the threads of a data-parallelism application more or less do the same work. However, it may not be appropriate to allocate equal time-quantum to all the threads of an application that uses pipelined parallelism because resource usage of the threads from different pipeline stages may differ greatly.

However, our experiments with different pipeline parallel applications reveal that allocating equal time-quantum to all the threads also works well for pipeline parallel applications. This is because the scaling-factor is calculated based on the resource usage of the entire application, which allows to account the overall effect of all the threads or the dominating pipeline stage threads. For example, consider 'ferret' pipeline parallel application from PARSEC benchmark. *ferret* is a search engine which finds a set of images similar to a query image by analyzing their contents. The program is divided into six pipeline stages -- the results of processing in one stage are passed on to the next stage. The stages are: Load, Segment, Extract, Vector, Rank, and Out. The speedup of ferret increases linearly starting from 6 threads to all the way up to 63 threads even though only 24 cores are available. The reason for the observed behavior is as follows. The *Rank* stage performs most of the work and thus the speedup of the application is determined by the *Rank* stage. Moreover the other stages perform relatively little work and thus their threads together use only a fraction of the compute power of the available cores. Thus, as long as cores are not sufficiently

utilized, more speedup can be obtained by creating additional threads for the *Rank* stage. Therefore, Rank stage threads of ferret program dominates the behavior of all other threads and represents resource usage of whole ferret program. Thus, since the scaling-factor represents the resource usage of the entire application, allocating time-quantum based on the scaling-factor works well also for pipeline parallel programs.

D. Implementation of FF policy

There are two important components of the implementation of FF policy framework: (1) providing same priority to all the application threads, and (2) allocating appropriate time-quantum based on the resource usage of the application. OpenSolaris provides a scheduling class called Fixed Priority scheduling [3]; with the combination of this class and priocntl(1) [4] utility, we can allocate same priority to all the threads of an application. However, there is no way to find appropriate time-quantum for an application in OpenSolaris with the fixed priority scheduling class. Moreover, this class does not provide any capability for updating time-quantum [3]. Thus, there is no way to deal with the phase changes of an application. Therefore, in addition to developing a scaling-factor table, we also perform continuously monitoring of an application to allocate appropriate time-quantum according to its phase changes.

Let us consider the FF policy implementation in detail. As shown in Algorithm 1, our implementation uses a daemon thread. First we start the target program with the default TS policy and start monitoring the program's last-level cache miss-ratio and lock-contention after the creation of target program's worker threads. We use cputrack(1) utility to monitor miss-ratio and prstat(1) utility for lock-contention with one second interval. We used a timer that fires a timer signal for every one second and the framework catches the signal and collects miss-ratio and lock-contention of the target program with one second interval, calculates a scaling-factor, and based on this it allocates appropriate time-quantum to the application threads using the scaling-factor table. More specifically, the framework measures a scaling-factor of the target application for every one second, and checks whether to change the time-quantum or not by comparing the current scaling-factor with the previous one. Although we can use an interval with milliseconds resolution, we used one second interval because our experiments showed that one second interval is enough to deal with the phase changes of the programs studied in this work. Although, one second time-interval is the minimum timeout value we could have used with the default implementation of prstat(1) utility, we modified this utility to allow time intervals with millisecond resolution to monitor lock-contention. Therefore, it is easy to use an interval less than one second for an application that experiences rapid phase changes.

Thus, our framework continuously monitors the target multithreaded program and allocates same priority using priocntl(1) utility and assigns appropriate time-quantum based on the scaling-factor table. Moreover, the overhead of this framework is negligible (0.02% of CPU utilization) and it requires no changes to the application source code or to the OS kernel.

Algorithm 1: FF Policy Framework

```
Profile Data Structure and Variables;
  Profile P:
    missRatio: last-level cache misses/accesses;
    lockContention: (% lock-contention/100);
  // range of the scaling-factor
  range = 0.10;
Subroutines:
  getProfile():
    Monitor missRatio using cputrack(1) and lockConetntion
    using prstat(1) of the target program with one second
    interval and return a Profile P;
  getScalingFactor(missRatio, lockContention):
    return [1 - max(missRatio, lockContention)] ;
  getTimeQuantum(scalingFactor):
    return corresponding TQ from the Scaling-Factor Table;

Input  : Target Multithreaded Benchmark Program
Output : Apply FF policy.

Start the target program with TS policy;
while program hasn't create its worker threads do
  | Sleep(); // checks like a deamon process
end

Wait for one more second to allow the application threads for
their initialization period;

oldP = getProfile();
oldScalingFactor = getScalingFactor(oldP.missRatio,
oldP.lockContention);
oldTQ = getTimeQuantum(oldScalingFactor);

Allocate oldTQ and same priority using priocntl(1) utility;

// continuous monitoring
repeat
  | newP = getProfile();
  | newScalingFactor = getScalingFactor(newP.missRatio,
  | newP.lockContention);
  | if (newScalingFactor > (oldScalingFactor + range)) or
  | (newScalingFactor < (oldScalingFactor - range)) then
  |   | newTQ = getTimeQuantum(newScalingFactor);
  |   | oldScalingFactor = newScalingFactor;
  |   | Allocate newTQ using priocntl utility;
  | end
until completion of the target program;
```

IV. Experimental Setup

This section describes the execution environment where FF policy is developed and evaluated.

A. Target Machine and OS

Our experimental setup consists of a Dell PowerEdge R905 server whose configuration is shown in Table II. As we can see this machine has 24 cores and is running OpenSolaris.

Table II: Target Machine and Operating System.

Dell™ PowerEdge R905:
24 Cores:
4 × 6-Core 64-bit AMD Opteron 8431 Processors (2.4 GHz);
L1 : 128 KB; Private to a core; L2 : 512 KB; Private to a core;
L3 : 6144 KB; Shared among 6 cores; Memory: 32 GB RAM;
Operating System: OpenSolaris.2009.06.

B. Benchmarks

We evaluate FF policy with a wide variety of benchmarks -- 22 benchmark programs in all. We also included a micro-benchmark [1] to study how FF policy works under varying levels of contention. This benchmark consists of M threads running on N cores that repeatedly acquire and release a single global lock. The critical section consists of a single call to gethrtime(), which takes around 300 ns to execute on our machine. Between lock acquires, threads busy-wait a fixed period of time before the first measurement and stop after the last one. Threads increment a local counter with each lock releases, and the benchmark harness computes throughput by comparing two successive readings of each thread's counter while threads continue to run.

The other 21 complete programs are as follows: eight programs (*streamcluster, facesim, canneal, x264, fluidanimate, swaptions, ferret, and bodytrack*) from PARSEC [26], 11 programs (*swim, wupwise, equake, gafort, art, apsi, ammp, applu, fma3d, galgel, and mgrid*) from SPEC OMP [27], *SPECjbb2005* [27], and *TATP* [24] database transaction program. The implementations of PARSEC programs are based upon *pthreads* and we ran them using *native inputs*. SPEC OMP programs were run on medium input data sets. SPECjbb2005 with single JVM is used in all our experiments. TATP (a.k.a NDBB and TM-1) uses a 10000 subscriber dataset of size 20MB with a solidDB [25] engine. TATP is not IO-intensive and disk performance does not affect it significantly [1]. In this work, we ran each experiment 10 times and present average results from the ten runs.

V. Evaluating FF policy

In this section, we analyze the effectiveness of FF policy using the microbenchmark and the 21 complete programs introduced in Section IV.

A. Against varying contention levels

Since FF policy completely avoids BS and specifically lock-holder thread preemptions, it is very effective against varying lock-contention levels. Fig. 6 demonstrates this. We use a microbenchmark where threads contend for a single global lock, with a fixed delay between requests [1]. High contention occurs for short requests on the left of the x-axis and drops off moving toward the right. We consider three cases, where the machine is 95% loaded (i.e., 23 threads), 150% loaded (i.e., 36 threads) and 200% loaded (i.e., 48 threads) [1]. As we move right along the x-axis, contention decreases, and throughput is improved in all three cases. As we can see in Fig. 6, when contention is high and the system is overloaded, program experiences high BS, and leads to poor performance. For lightly loaded systems, FF performs slightly better than TS because program experiences low BS. However, overall, FF outperforms TS significantly at all contention levels.

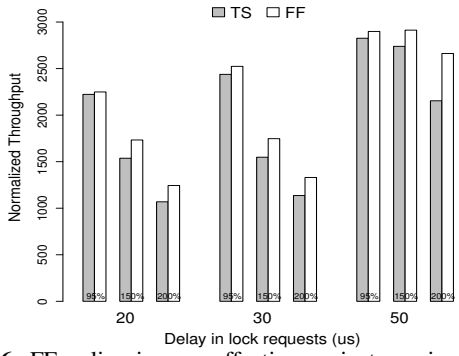


Figure 6: FF policy is very effective against varying contention levels.

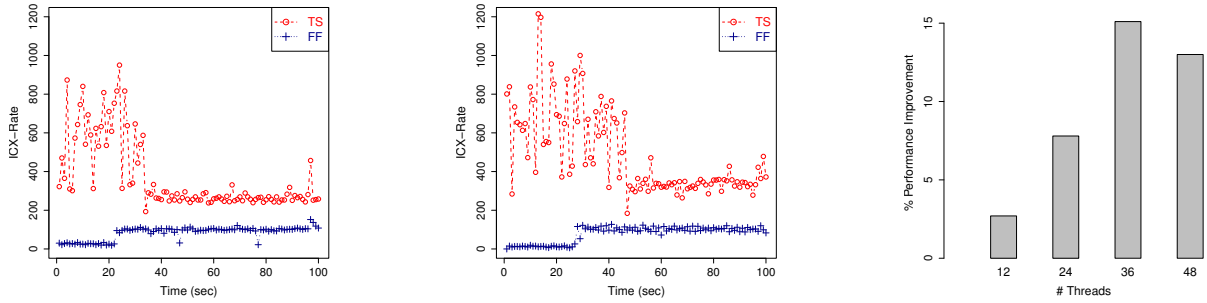
B. Against phase changes

As explained in Section III-D, the FF framework continuously monitors the target multithreaded program and allocates appropriate time-quantum to effectively deal with its phase changes. For example, consider the *ammp* program which exhibits two significantly different execution phases described in Section III-B. Using the scaling-factor table, the FF policy allocates appropriate time-quantum according to the resource usage of its phases. As *ammp* suffers from high lock contention for around 84% of elapsed time in the first phase, scaling-factor is 0.16 for the first phase. Here lock-contention is higher than miss-ratio value. Likewise, scaling-factor is 0.88 for the second phase of the *ammp* program as it suffers from low lock contention for around 12%. Therefore, using continuous monitoring, the FF policy allocates time-quantum 200 ms for the first phase, 20 ms for the second phase, and thus effectively deals with the phase changes of the *ammp* program.

As shown in Fig. 7, FF policy is very efficient against the phase changes of *ammp* program. It dramatically reduces ICX-Rate and leads to high performance. As we can see in Fig. 7(c), *ammp* achieves up to 15% performance improvement with FF policy. Since, we use small time-quantum for the phases that have high scaling-factor, we can expect a little increase in TQE ICX. Thus, as we can see in Fig. 7, there is a rise in the ICX-Rate in the second phase with FF policy. However, FF policy produces less TQE ICX compared to TS policy at both 100% and 150% loads.

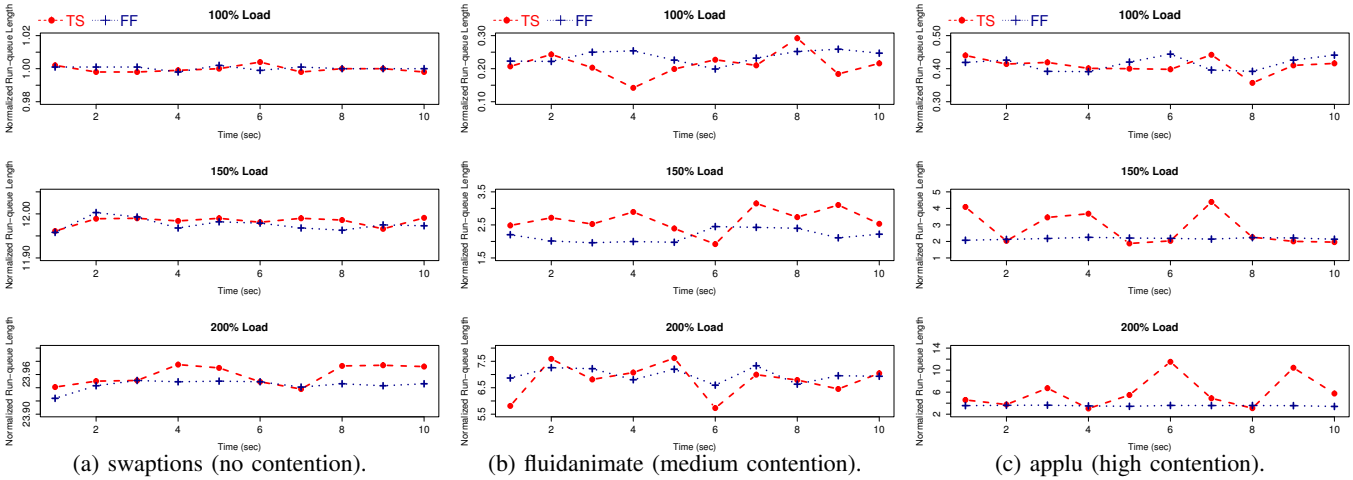
C. Against dynamic load changes

Since FF policy completely eliminates BS, consequently reducing CX-Rate, it brings stability in load management. Fig. 8 demonstrates this. The y-axis of the figure represents normalized run-queue length of the system, i.e. total number of runnable threads on the dispatcher queues of the system [3], [4]. The x-axis shows the time in seconds. Fig. 8 shows the normalized run-queue lengths of swaptions, fluidanimate, applu programs at 100%, 150%, and 200% loads. As shown



(a) Dramatic reduction in ICX-Rate at 100% load. (b) Dramatic reduction in ICX-Rate at 150% load. (c) Performance improvement over TS policy.

Figure 7: FF policy effectively deals with phases of ammp program and improves its performance.



(a) swaptions (no contention). (b) fluidanimate (medium contention). (c) applu (high contention).

Figure 8: FF policy avoids spikes in the load.

in Fig. 8(a), there are no significant load changes with both TS and FF policies in case of very low contention swaptions program even at 200% load. However, there are significant spikes in the load for high contention programs -- fluidanimate and applu -- with TS policy, but there are no spikes in the load with FF policy. Therefore, by completely eliminating BS and consequently reducing CX-Rate, FF policy avoids spikes in the load and leads to high performance. Moreover, threads experience higher CPU latencies with TS policy under high loads compared with FF policy, i.e., threads wait for longer times in the dispatch queues with TS policy, which slows down the progress of the application.

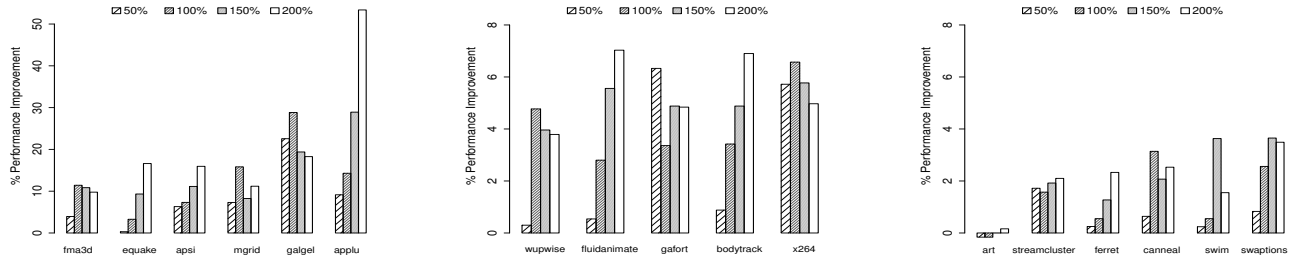
Thus, FF policy is agnostic to dynamic load changes and improves performance predictability of multithreaded programs running on multicore machines. In contrast to this, the load-controller [1] is sensitive to spikes in the load.

D. Performance Improvements

As shown in Fig. 9 and 10, FF policy improves performance for a wide variety of programs at 50%, 100%, 150%, and 200% loads over TS policy. As high contention programs suffer heavily from BS, they achieve tremendous performance improvement with FF policy. Fig. 9(a), Fig. 10, and Fig. 7(c) all show this. There are moderate improvements for the medium contention programs shown in Fig. 9(b) and small improvements for the low contention programs shown in Fig. 9(c). Although FF policy considers whole application for allocating time-quantum, as shown in Fig. 9(b) and Fig. 9(c), FF policy improves performance of *pipeline parallel* programs bodytrack, x264, and ferret.

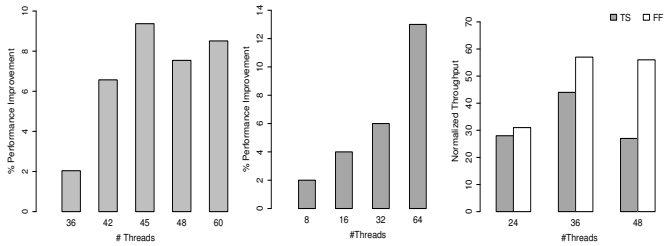
More specifically, at 100% load, FF policy achieves more than 10% performance improvement for five programs with a maximum of 35% improvement, 4%-10% for six programs, less than 4% for nine programs, and there is no improvement for one program over TS policy. At 200% load, FF policy achieves more than 10% performance improvement for eight programs with a maximum of 107% improvement, 4%-10% for six programs, less than 4% for seven programs over TS policy. Moreover, FF policy also achieves performance improvements for several programs under light loads, specifically at 50% load.

Since our execution environment is different from [1], it is not possible to directly compare the performance improvement data of TATP using our FF policy against the performance improvement with the load-controller [1]. However, as shown in Fig. 10 (c), FF policy improves performance of TATP like the load-controller does and also the performance degradation is steady as load increases. Moreover, in contrast to the load-controller, we did not need to modify the application source code for ensuring visible spin locks and also FF policy is agnostic to dynamic load changes.



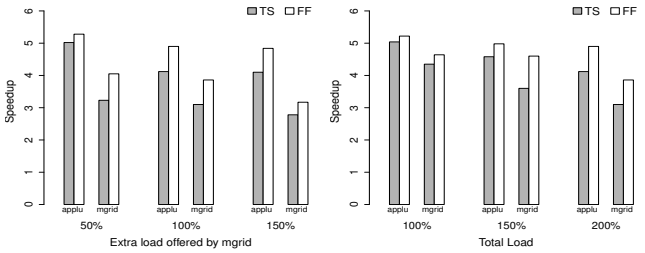
(a) High Performance Improvements. (b) Medium Performance Improvements. (c) Low Performance Improvements.

Figure 9: FF policy improves performance of a wide variety of programs.



(a) SPECjbb2005. (b) facesim. (c) TATP

Figure 10: Performance improvement of SPECjbb2005, facesim, and TATP with FF policy. SPECjbb2005 creates 35 threads with one warehouse.



(a) applu with extra load offered by mgrid. (b) Both applu and mgrid offer same load.

Figure 11: FF policy is very effective against parallel runs of more than one application.

E. Discussion

The previous section demonstrates that FF policy improves performance of a wide variety of multithreaded applications at different loads. We next discuss some of the extensions and limitations of FF policy framework.

1) Concurrent runs of more than one application

TS policy has been widely used in modern operating systems. It does not consider the whole application but rather assigns priority and time-quantum on a per thread basis. That is why FF policy significantly outperforms TS policy when single multithreaded application is running on the system. However, TS policy is quite effective when there are multiple multithreaded applications running on a multicore system. Therefore, we would like to see how FF policy works with parallel runs of more than one application on a multicore system. For this, we conducted two experiments. In the first experiment we ran applu with 24 threads along with extra load offered by mgrid – we ran 24 threads of applu along with 12 threads, 24 threads, and 36 threads of mgrid. In the second experiment we run both applu and mgrid with equal number of threads – (12, 12), (18, 18), and (24, 24) threads. As shown in Fig. 11 (a) and Fig. 11 (b), FF policy greatly outperforms TS policy. Similar performance improvements resulted from several experiments of running multiple applications with FF policy. We are unable to present those due to lack of space. Thus, FF policy is also effective when there is more than one application running on the system.

2) Limitations

Although FF policy is quite effective against pipeline parallel programs, still we can improve their performance if we use different and appropriate time-quanta for different pipeline stage threads according to their resource usage. The 22 applications studied in this work mainly stress CPU and Memory. However, it is easy to extend FF policy framework for IO-intensive applications by considering how much time threads utilizing CPU, IO characteristics, and modifying the scaling-factor table appropriately.

VI. Related Work

The problems with spinning and blocking are well known and have prompted many approaches, such as queue-based spinlocks [7], [8] and ticket spinlocks [9], to alleviate these problems. Both Queue-based spinlocks and ticket spinlocks provide an efficient way of orderly lock-handoffs because waiting threads form a FIFO queue and each lock handoff targets a specific thread. However, they also suffer from lock-holder thread preemptions at high load and create lock convoys [1], [10]. Time-published locks [11] eliminate the main problem with queue-based locks by only handing the lock to running threads. However, these also allow lock holders to be vulnerable to preemption [1]. By limiting the number of waiting threads which can respond simultaneously, backoff-based techniques [12], [13], [19] provide another solution to the "thundering herd" problem [1], where all waiting threads race for the lock at each release and cause both contention and memory traffic. However, finding optimal backoff length for the general case is a challenging problem. Hybrid spin-then-block techniques [3], [17], [19] use spinning to reduce context switching imposed by a blocking primitive. However, these also face challenges to provide optimal balance between spinning and blocking as load increases [2].

In order to avoid unexpected load changes because of the interactions between irregular parallelism of database

applications and scheduling, several admission control techniques [14], [15], [18] are employed. These techniques monitor system statistics regarding CPU, memory, lock contention and tune the amount of work allowed into the system [1]. Using simulations, Gupta et al. [12] explored the trade-offs between the use of busy-waiting and blocking synchronization primitives and their interactions with the scheduling strategies. They also explored the impact of the scheduling strategies on the caching behavior of the applications. Several researchers [20]–[23] use application characteristics such as cache miss ratio to make better scheduling decisions in multicore environments.

To alleviate the problems with the hybrid spin-then-block approaches, Johnson et al., [1], [6] proposed a "load control" mechanism that decouples load management with the contention management. This approach uses blocking to control the number of runnable threads and then spinning in response to contention. Although this approach works well, it needs to modify the applications for providing visible spin lock and load controller is sensitive to large changes in the load. It also faces problems when priority inversions arise due to nested critical sections, it does not completely eliminate BS, and leads to high CX-Rate as load increases. Moreover, the implementation of load controller uses 7 ms as an update interval, with which, it is difficult to obtain accurate processor usage statistics and the overhead increases linearly with the number of threads [1].

In contrast, in this work, we dramatically reduce the problems caused by the unwanted interactions between OS scheduling policy and contention management policy. We presented a scheduling policy called faithful scheduling (FF), where all threads of an application have same priority for its entire execution, however the time-quantum is allocated according to their usage of system resources. By providing same priority to all the threads of an application, this policy completely eliminates BS, dramatically reducing CX-Rate, and leads to high performance. It avoids priority inversion problems and therefore it is not effected by nested critical sections. Moreover, FF policy avoids spikes in the load, it does not require any modifications to application source code, and its implementation is also simple with negligible overhead.

VII. Conclusions

This paper presents a scheduling policy called Faithful (FF) Scheduling, where threads of an application have same priority and the time-quantum is allocated according to the resource usage of the entire application. FF policy is very effective against varying contention levels, phase changes, dynamic load changes, and improves performance of a wide variety of benchmark programs over the default scheduling policy Time Share (TS). Moreover, FF policy is an attractive approach as it does not require any changes to the application source code or the OS kernel.

Acknowledgements

The authors would like to thank Antoni Wolski and Simo Neuvonen of IBM Research for their help in installing the TATP benchmark and Ryan Johnson of University of Toronto for his help in developing the micro benchmark. Jim Mauro and Rick Weisner of Oracle for their help throughout this work. The authors would also like to thank the anonymous reviewers for their helpful comments.

This work is supported in part by NSF grants CCF-0963996 and CNS-0810906 to the University of California, Riverside.

References

[1] R. Johnson, R. Stoica, A. Ailamaki, T. C. Mowry. Decoupling contention management from scheduling. In *ASPLOS* 2010.

[2] L. Boguslavsky, K. Harzallah, A. Kreinen, K. Sevcik, A. Vainshtein. Optimal strategies for spinning and blocking. *Journal of Parallel and Distributed Computing*, Volume 21, Issue 2, PP. 246-254, May 1994.

[3] R. McDougall, J. Mauro. Solaris Internals, Prentice Hall Publications, Second Edition, July 2006 .

[4] R. McDougall, J. Mauro, and B. Gregg. Solaris Performance and Tools: DTrace and MDB Techniques for Solaris 10 and OpenSolaris, Prentice Hall, 2006.

[5] B. Cantrill, M. Shapiro, and A. Leventhal. Dynamic instrumentation of production systems. In *USENIX ATC*, 2004.

[6] R. Johnson, M. Athannassoulis, R. Stoica, and A. Ailamaki. A new look at the roles of spinning and blocking. In ACM SIGMOD *DaMoN* workshop, Providence, RI, July 2009.

[7] J. M. Mellor-Crummey, M. L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, ACM TOCS 9,1, p.21-65, Feb. 1991.

[8] P. Magnussen, A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. In *International Symposium on Parallel Processing*, pp. 165-171, Apr. 1994.

[9] D. P. Reed and R. K. Kanodia. Synchronization with Event-counts and Sequencers. Communications of the ACM, 22(2):115-123, Feb. 1979.

[10] M. Blasgen, J. Gray, M. Mitoma, and T. Price. The convoy phenomenon. ACM SIGOPS Operating Systems Review 13,2, pp. 20-25. 1979.

[11] B. He, W. N. Scherer III, and M. L. Scott. Preemption adaptivity in time-published queue-based spin locks. In *HiPC*, 2005.

[12] A. Gupta, A. Tucker, and S. Urushibara. The Impact Of Operating System Scheduling Policies And Synchronization Methods On Performance Of Parallel Applications. *SIGMETRICS Perform. Eval. Rev.*, 1995.

[13] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *ISCA*, pp. 396-406, 1989.

[14] N. Bartolini, G. Bongiovanni, Simone Silvestri. Self-*through self-learning: Overload control for distributed web systems. In *International Journal of Computer and Telecommunications Networking* 53,5, pp. 727-743, Apr. 2009.

[15] M. Carey, S. Krishnamurthi, and M. Livny. Load control for locking: the "half-and-half" approach. In *PODS*, Apr. 1990.

[16] J. Carlstrom and R. Rom. Application aware admission control and scheduling in web servers. In *INFOCOM*, 2002.

[17] H. Franke, R. Russell, M. K. Fuss. Futexes and furwocks: Fast userlevel locking in linux. In Proc. 2002 Ottawa Linux Summit, 2002.

[18] A. Monkeberg, G. Weikum. Performance evaluation of an adaptive and robust load control method for the avoidance of data contention thrashing. In *VLDB*, 1992.

[19] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *Dist. Computing Systems*, 1982.

[20] E. W. Parsons, K. C. Sevcik. Benefits of speedup knowledge in memory-constrained multiprocessor scheduling. *Performance Evaluation* 27/28, pages 253-272, 1996.

[21] M. Bhadauria, S. A. McKee. An Approach to Resource-Aware Co-Scheduling for CMPs. In *ICS*, June, 2010.

[22] Guangdeng Liao, Danhua Guo, Laxmi N. Bhuyan and Steve R. King. Software Techniques to Improve Virtualized I/O on Multi-core Platforms. In proceedings of ANCS 2008, San Jose, USA, 2008.

[23] S. Zhuravlev, S. Blagodurov and A. Fedorova. Addressing Shared Resource Contention in Multicore Processors via Scheduling. In *ASPLOS*, 2010.

[24] IBM. Telecom Application Transaction Processing (TATP) Benchmark Description. Available online at http://tatpbench-mark.sourceforge.net/TATP_Description.pdf.

[25] IBM solidDB 6.5 Fix Pack 3 - 6.5.0.3 Build 2010-10-04 https://www-304.ibm.com/support/docview.wss?uid=swg24028071

[26] C. Bienia, S. Kumar, J.P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT*, 2008.

[27] SPEC OMP, SPECJbb2005. http://www.spec.org/