Architectural Support for Handling Jitter in Shared Memory Based Parallel Applications

Sandeep Chandran, Prathmesh Kallurkar, Parul Gupta, *Senior Member, IEEE*, and Smruti R. Sarangi, *Member, IEEE*

Abstract—With an increasing number of cores per chip, it is becoming harder to guarantee optimal performance for parallel shared memory applications due to interference caused by kernel threads, interrupts, bus contention, and temperature management schemes (referred to as *jitter*). We demonstrate that the performance of parallel programs gets reduced (up to 35.22 percent) in large CMP based systems. In this paper, we characterize the jitter for large multi-core processors, and evaluate the loss in performance. We propose a novel jitter measurement unit that uses a distributed protocol to keep track of the number of wasted cycles. Subsequently, we try to compensate for jitter by using DVFS across a region of timing critical instructions called a *frame*. Additionally, we propose an OS cache that intelligently manages the OS cache lines to reduce memory interference. By performing detailed cycle accurate simulations, we show that we are able to execute a suite of Splash2 and Parsec benchmarks with a deterministic timing overhead limited to 2 percent for 14 out of 17 benchmarks with modest DVFS factors. We reduce the overall jitter by an average 13.5 percent for Splash2 and 6.4 percent for Parsec. The area overhead of our scheme is limited to 1 percent.

Index Terms—CMP, hardware support for OS, DVFS, operating system jitter, HPC application

1 Introduction

The number of cores per chip are doubling roughly every two years as predicted by Moore's law. Consequently, traditional high performance computing (HPC) applications are increasingly being ported to CMPs [1], [2]. As the number of cores on a CMP scales beyond 16 or 32, HPC applications will start becoming extremely sensitive to the length of sequential portions and critical sections in the code. This is a direct consequence of Amdahl's law. Hence, it will become necessary to properly tune the CMP systems (both HW and SW) akin to HPC clusters such that optimal performance can be guaranteed in the face of *jitter* inducing events such as system calls, interrupts, kernel threads, system events, daemons, and other processes. A small amount of jitter in a critical section can elongate the critical path and can lead to a disproportionate amount of slowdown.

Prior work has mostly focused on managing jitter for large clusters [3], [4], [5], [6]. However, we could not find any prior studies that studied the effect of jitter on CMP based shared memory applications. In this paper, we study the impact of jitter on a 16 core shared memory CMP using POSIX thread(pthread) based benchmarks. We observed slowdowns of up to 41 and 27 percent (see Fig. 10) in the Splash and Parsec benchmark suites respectively.

Consequently, in this paper we *exclusively focus on reducing jitter for general purpose non-real time HPC applications.* We shall focus on parallel real time applications with possibly inviolable hard deadlines in the future. The main sources of jitter [3], [7] are OS induced jitter, multi-threading/tasking, and cpu events. We further subdivide the OS jitter into two types—active and passive. Timer interrupts and I/O interrupts that are delivered by external agents contribute to *active* OS jitter, whereas jitter caused by system calls made by target applications contribute to *passive* jitter. We demonstrate in Section 6.1.1 that passive synchronization jitter caused by pthread based system calls to enter and exit critical sections/barriers accounts for about 90 percent of the total jitter in a properly tuned system (defined in Section 5). However, pthread based synchronization calls are integral to a shared memory based HPC system, and to the best of our knowledge, prior work has not looked at it in great detail. A properly tuned system adopts solutions already devised by the HPC community to minimize jitter such as real time kernels, threads with real time priority, interrupt isolation, and curtailing all forms of extraneous activity. For example, it is easy to minimize jitter due to other processes by running a system in Linux single user mode and setting thread priorities to real time. Likewise, cpu based power and thermal events such as voltage frequency scaling can be switched off for the duration of execution of an HPC task, or we can use superior cooling methods.

In this paper, we focus most of our effort in trying to reduce the jitter due to pthread based synchronization calls. We propose a novel piece of hardware called the *jitter unit*. It runs a distributed protocol to estimate the number of cycles/second lost due to OS jitter and a host of other events including processor events, and timer interrupts. The jitter unit consists of a set of intelligent counters to measure jitter related events that take cues from special instructions inserted into the standard POSIX thread library and the

- *S. Chandran, P. Kallurkar, and S.R. Sarangi are with the Department of Computer Science and Engineering, Indian Institute of Technology Delhi, New Delhi 110016, India.*
- *P. Gupta is with IBM Research Labs, India.*

Manuscript received 20 July 2012; revised 11 Mar. 2013; accepted 17 Apr. 2013; date of publication 23 Apr. 2013; date of current version 21 Mar. 2014.
Recommended for acceptance by K. Li.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2013.127

1045-9219 © 2013 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

kernel. We envision this unit to be a non-intrusive monitoring mechanism, which does not change or interfere with the normal operation of the processor in any way.

We start out by dividing the total execution into discrete quanta of dynamic instructions called *frames*. For regular HPC applications, the entire program is a single frame. However, for parallel real time applications such as software radio [8], a frame can correspond to the code that processes a single packet. For example, in the WiMax [9] protocol, we need to process a given packet in less than 5 milliseconds. Thus, a frame in a WiMax application can be defined to be 5 milliseconds long. We further divide a frame into a set of *subframes*, where each subframe is n seconds long. n is typically between 200 μs to 1 millisecond. In each subframe, we estimate the amount of jitter by reading the values saved in different jitter units, and add it to the accumulated jitter within the current frame. We try to compensate for this accumulated jitter over the next few subframes by modulating their voltage and frequency.

We observe that using DVFS alone with inputs from the jitter unit is not sufficient to curtail jitter. Hence, we propose to supplement the scheme with a small OS cache. This is a 64 KB cache at the L2 level. It is meant to hold the cache lines that belong to the operating system. Moreover, the OS cache and the regular L2 cache can share cache lines between them to reduce conflict and capacity misses. However, there are some subtle issues in the design of an OS cache namely shared data (between user level processes and the kernel), and cache coherence. We shall delve into these issues in Section 4.2.

We present the background and related work in Section 2, characterize synchronization jitter in Section 3, show the implementation of the jitter unit in Section 4, present the evaluation setup in Section 5, display the results in Section 6, and finally conclude in Section 7.

2 Background and Related Work

2.1 Definitions

Time sensitive task. A task (serial/parallel), which is being monitored for jitter by our system.

Definition of jitter. Let us consider a sequence, $\mathcal{S}$, of dynamic instructions belonging to a time sensitive task. Let it take time t to execute on an ideal machine, and time t' on a non-ideal machine. The jitter J is defined as $t' - t$. An *ideal* machine is defined as a system with zero interference from any external source.

We note that $\mathcal{S}$ may contain interrupts to the kernel, and may consist of disruptions introduced by multi-threaded code. Our definition of jitter, which is similar to that defined in [10], takes into account sources of delay other than the OS.

2.2 Sources of Jitter

According to De et al. [3], [7] there are four main sources of jitter in a computer system namely OS activity, multiple threads (SMT interference), power/thermal management, and the hypervisor. We do not consider hypervisors in this work. A detailed description of the sources of jitter can be found in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2013.127. We shall provide a brief summary in this section.

Jitter can be primarily divided into two types—*active* and *passive*. Active jitter is defined as jitter caused by external events such as interrupts. Prior studies [3] have found the timer interrupt to be the largest contributor of active jitter (up to 85 percent). I/O and network activity have been found to account for the rest. The reader should note that it is not the case that timer interrupts take a long time to get processed. The kernel opportunistically uses the timer interrupts to schedule its own work.

Passive jitter is caused as a side effect of system calls. In parallel benchmarks, synchronization calls (mutex lock, unlock, etc.) often lead to system calls. The kernel uses these opportunities to schedule its own work to run book keeping tasks, daemons, or bottom-halves of device drivers. In our studies we have found synchronization interrupts to be more frequent than timer interrupts. Hence, most of the jitter is accounted for by synchronization interrupts.

It is possible to reduce jitter by forcing the interrupts to be handled on a fixed set of cores (cpu isolation) or using proprietary real time operating systems. The former is a part of our baseline system, whereas, the latter has prohibitive performance overheads. Jitter can also be caused by multiple threads, and power/temperature management events.

3 Characterization and Detection of Synchronization Jitter

3.1 Basics of POSIX Threads

Fig. 1 shows the different types of synchronization operations in the POSIX threads (pthreads) library. Lock-Unlock start and end a critical section using a memory address as the lock address. Signal-Wait and Broadcast-Wait are two paradigms in which one thread waits for another thread to signal it to resume. The difference between signal and broadcast is that signal is one-to-one communication, and broadcast is one-to-many communication. The latest version of the POSIX thread library has a barrier primitive. Since it internally uses the broadcast mechanism, we omit it for the sake of brevity. There are three more primitives for thread creation and termination—create, exit, and join. In the *join* operation, one thread waits for another thread to finish. We define a set of events that are fired when we enter a synchronization library call, and exit it. They are shown in Table 1. Let us now look at typical communication scenarios for measuring signal-wait jitter. Other cases can be handled similarly.

3.2 Scenarios for Signal-Wait Synchronization

3.2.1 Case 1

In Fig. 2, we look at a typical scenario for a signal-wait communication pattern. First, a thread on core 1 issues a wait_entry (w_e) call. We envision a dedicated piece of hardware called the *jitter unit* on each core. The jitter unit on core 1 makes an entry of this by logging it in a dedicated wait buffer. Subsequently, a thread on core 2 tries to signal the waiting thread. The jitter unit on core 2 catches this event, and broadcasts the s_e event to the other jitter units.

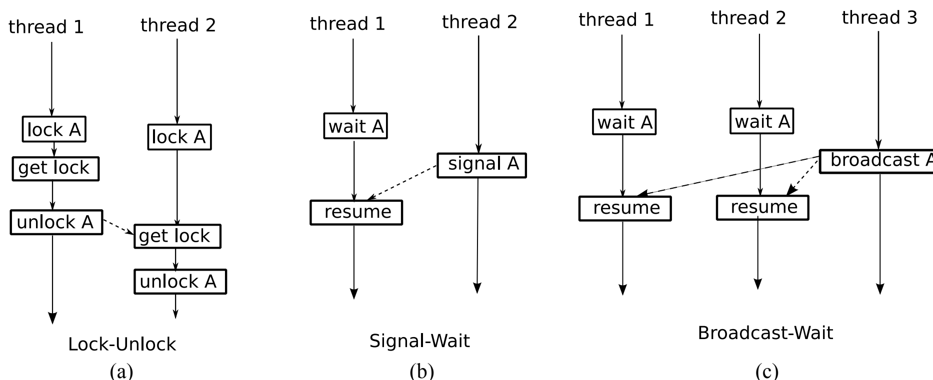

Fig. 1. Synchronization primitives in the pthread library.

The s_e event contains the lock address, and the id of the thread that is going to be woken up (the pthread library can compute the id of the thread to be woken up very quickly without any system calls). We don't expect the overhead to be more than a couple of cycles. Once core 1 receives the s_e event, it starts searching for entries in its wait buffer that match the address and the thread id. If a match is found, then a thread is going to be woken up in the near future. The jitter unit timestamps the wait entry.

In parallel, the pthread code on core 2 typically does some pre-processing, and then sends an interrupt to the kernel (int 0×80 instruction on x86). The kernel immediately runs the schedule function, in which the kernel can either send an inter-processor-interrupt (ipi) to core 1 to resume the waiting thread, or it can mark the waiting thread as ready, and just return. Fig. 2 shows the former case. The latter case is treated the same way.

The time between s_e and ipi on core 2 should be within limits. If it is not the case, then this means that there is some jitter in the kernel. After core 1 receives the ipi, it immediately transitions to kernel mode. Let us assume that the waiting thread has the highest priority. Then the kernel will exit (k_x) and wakeup the thread firing the w_x event.

The jitter in core 1 is the time difference between s_e and w_x in Fig. 2. We call it signal-wait jitter. The justification for this reasoning is as follows. From the programmer's point of view, the point of signal entry, s_e, is when she expects core 1 to start instantaneously. She further expects the signal call to finish instantaneously. We need to take into account a certain base value for any kind of synchronization operation. We typically assign N μs for every operation. If the time, T, exceeds that, then the jitter is $T - N$. Given this reasoning, if the time between the receipt of the event s_e on core 1 and w_x (wait exit) on core 1 differ significantly, then we can infer the existence of OS jitter.

Likewise on core 2, after sending the ipi, the kernel running on core 2 can schedule other tasks like daemons/interrupt handlers. The time difference between s_e and s_x is accounted for as purely signal jitter on core 2.

TABLE 1
List of Events

event	symbol
lock entry	l_e
lock exit	l_x
unlock entry	u_e
unlock exit	u_x
signal entry	s_e
signal exit	s_x
broadcast entry	b_e
broadcast exit	b_x
wait entry	w_e
wait exit	w_x

3.2.2 Case 2

Core 1 might wakeup another thread after receiving the ipi, and then wake the time sensitive thread (details in Appendix B, available in the online supplemental material).

3.2.3 Case 3

It is possible that the time sensitive thread that was originally on core 1, wakes up on core 3 as shown in Fig. 3. In this case, core 3, will be aware of the fact that there has been a migration. It will broadcast the id of the thread, and get the time at which the corresponding signal event was issued from core 1 and calculate the jitter appropriately. (details in Appendix B, available in the online supplemental material).

We consider these cases *exhaustive* since we observe in our experiments that their coverage is more than 99.999 percent. Please note that it is possible to trivially extend our scheme to consider the existence of multiple time sensitive tasks. In this case, we need to have dedicated state in the

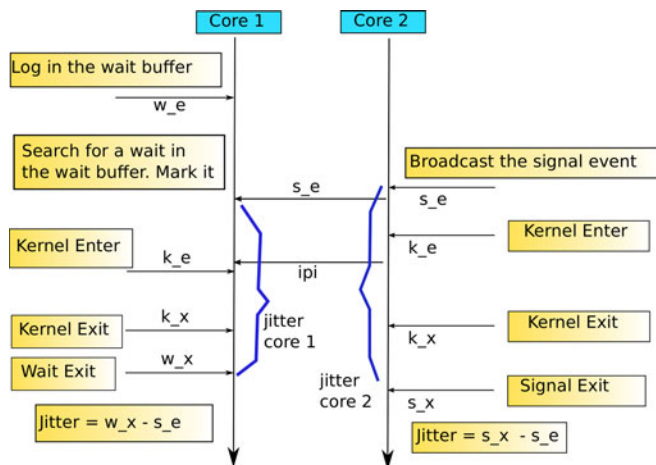

Fig. 2. Synchronization involving two cores.

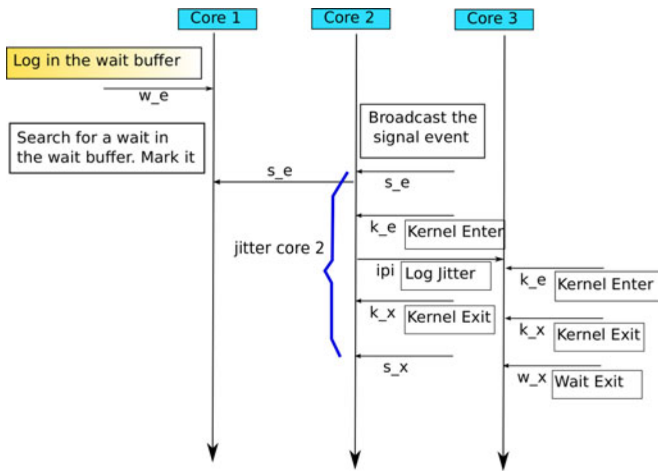

Fig. 3. Synchronization involving three cores.

jitter unit, and a wait queue for each task. Furthermore, each message needs to be stamped with the thread identifier.

4 Implementation Details

We propose two schemes to contain jitter. The first is a method to estimate the total jitter due to kernel interference. In this scheme, we use a dedicated piece of hardware called the *jitter unit* that runs a distributed protocol to calculate the amount of jitter experienced by a time sensitive task. Based on the amount of jitter, we try to compensate for it using voltage-frequency scaling.

The other approach is to use an OS cache at the L2 level to reduce the interference by the OS in the memory system. To further reduce conflict and capacity misses, we propose a method to seamlessly share lines between the OS cache and the regular L2 cache.

4.1 Jitter Unit

Every core has a jitter monitoring subsystem called the *jitter unit* as shown in Fig. 15 (Appendix C, available in the online supplemental material). The jitter unit will be periodically notified about different events by instructions in our modified pthread library, and events snooped from the processor and the bus.

4.1.1 Event Monitoring

A *time sensitive* application thread starts with letting the jitter unit know about itself by inserting its thread id in a model specific register, *jitter-reg* using an assembly instruction. This register is automatically unset when the processor switches to kernel mode or executes a pause, or halt instruction. When a time sensitive thread is swapped out, its current PC along with the thread id is recorded in the *thread-list* and broadcast to the rest of the jitter units in other cores. They also record the (PC, thread id) in their *thread-list*. The jitter units need to snoop the PC after a kernel exit event (k_x) and match the PC with values stored in their *thread-list*. If there is a match, then the jitter unit knows that it is a time sensitive thread with a given thread id, and monitoring jitter can proceed.

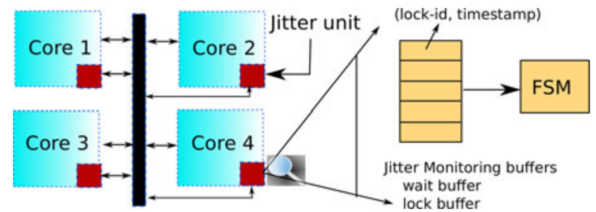

Fig. 4. High-level design of the jitter unit (One per core).

CPU events. CPUs have a lot of power management events like instruction throttling, reduction of frequency, powering down units, and so on. For every event, we calculate the estimated slowdown. We allow users to set this. For example, if we halve the frequency for 100 cycles, then we have roughly lost 100 cycles as per our definition of jitter.

Bus events. The bus arbiter will now monitor the bus to find out how many messages belonging to time sensitive tasks are getting delayed. We propose to use the scheme in [11] to measure the cycles lost.

4.1.2 OS Jitter Events

We instrument the POSIX thread (pthread) library (part of the standard C library) to track the following methods: create, exit, join, lock, unlock, signal, broadcast, and wait, as explained in Section 3 and Table 1. For each event, the pthread library writes the process id (pid), thread id (tid), event type (ev_type), and the memory address of the lock (if any), and time, to separate registers accessible by the jitter unit. We track two more events called kernel_entry (k_e) and kernel_exit (k_x). kernel_entry is fired when the kernel starts executing and likewise kernel_exit is fired when there is a context switch to a user process. We envision custom logic that can monitor the supervisor bit in processors to find out when the kernel is executing, and when it has stopped executing.

4.1.3 Design of the Jitter Unit

The detailed design of the jitter unit is shown in Appendix C, available in the online supplemental material. We summarize the main structures in this section. The high level design is shown in Fig. 4. The jitter unit is specific to each core and processes events sent by the core, and some events sent on the inter-core bus to compute the jitter experienced by the time sensitive thread.

The design of the jitter unit can broadly be divided into three parts: 1) information about the jitter experienced by the current thread (jitter state), 2) PCs of different threads in the time sensitive process, 3) events of interest that are used to compute jitter.

The jitter unit maintains information about the current thread, and especially the amount of jitter experienced in the current frame such that it can use this information to compensate for the resultant slowdown. This is known as the *jitter state* of the thread. Additionally, the jitter unit maintains information about the position of all the threads in a time sensitive process in terms of their program counter (PC) values before a context switch, in an SRAM array called a *thread list*. The thread list is used to initialize the jitter state of a core upon a thread migration.

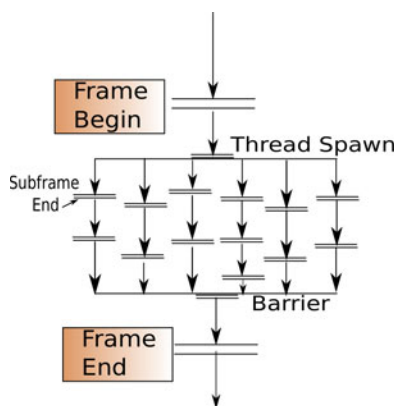

Fig. 5. Subframes in a multi-threaded code.

The most complicated part of the jitter unit contains the storage and logic to compute the jitter experienced by the time sensitive threads by logging events of interest. There are two main storage structures—lock buffer and wait buffer. The *lock buffer* is used to save the timestamp of lock events such that the jitter can be calculated once the corresponding unlock is issued. Likewise, the *wait buffer* is used to log wait events such that we can calculate the jitter for signal-wait and broadcast-wait synchronization patterns. Each jitter unit contains a finite state machine (FSM) to track the relationship between the different synchronization events and compute the relevant time intervals. The logic follows the patterns shown in Figs. 2 and 3. If these intervals exceed a pre-specified threshold, then the extra time is logged as jitter. Further details can be found in Section C.2.1 (It is there in the appendix).

4.1.4 Calculating the Critical Path

Fig. 5 shows a typical example of multi-threaded code where several threads are spawned simultaneously and join with a barrier. If a frame is wholly contained within a thread, then we try to reduce the time lost in jitter by applying DVFS across the subframes. The problem arises when a thread spawns other threads or threads get coalesced with a join operation.

To solve this, we force a subframe deadline when a thread is created and joined. For the newly created thread, we initialize its jitter counter with the jitter of the parent. When thread A finishes its execution and joins thread B, we set the jitter of thread B, to the maximum jitter of both threads. When n threads join the parent thread, the jitter-count is the maximum of the n threads and the parent. This procedure ensures that the jitter-count accurately reflects the critical path in a multi-threaded program.

A frame can thus span multiple threads. The programmer should ensure that it corresponds to a piece of computation that has a real time connotation. In the absence of programmer annotation, the jitter unit considers the entire span of program execution as a single frame. However, a subframe is defined for just one thread, and it has a specific DVFS setting. In our scheme, DVFS is applied on a per-core basis.

4.1.5 Jitter from Multiple Threads/Tasks

We observe that there is sometimes significant jitter introduced by kernel threads by displacing lines required by the time sensitive application in the cache. We observe that the penalty incurred in displacing L1 cache lines is not as high as the case of L2 lines. This is because of the high memory access latencies of the L2. Consequently, we propose to maintain a time sensitive, ts, bit for every L2 line. Whenever the kernel evicts a line with the ts bit on, we increment the $evicted_lines$ count for the subframe. After the end of the subframe, we multiply the number of evicted lines by the memory access time and then divide it by the number of banks to get an estimate of the jitter due to L2 cache line eviction. We add this value to the jitter-count for a subframe. We observe that this rough heuristic gives us acceptable results in our experiments.

4.1.6 Control

Based on the amount of jitter measured by the jitter unit residing on the core, a DVFS controller decides the DVFS multiplier for the next subframe. The decision is based on a lookup table maintained in software. Since, the DVFS controller itself is in software, it can be configured in different ways to mitigate jitter. The multipliers can be chosen very aggressively in which case, the power consumed may be too high; else a nominal setting may be used if the system needs to be optimized for power.

For each subframe, we record its CPI, and the L2 miss rate. The performance(insts/sec) is given by [12]:

$$P = \frac{f}{CPI_{comp} + mr * mp}. \tag{1}$$

Here, CPI_{comp} is the clock cycles per instruction barring L2 misses, mr is the L2 miss rate, and mp is the miss penalty in cycles. The L2 miss rates and the IPC remain more or less constant across a program phase [13], which is much longer than a subframe. Based on this information, we can get an estimate of the time the next subframe will take. Simultaneously, we maintain a count of the time lost to jitter using our measurement mechanisms.

Before the start of each subframe, we have two estimates—$\tau(f)$ and J. $\tau(f)$ is the expected time of execution for the subframe at a given frequency, and J is the time that has been currently lost to jitter. It is saved in the *jitter-count* register. Let f_0 be the nominal frequency of the machine. We set f such that $\tau(f) = \tau(f_0) - J$. At the end of the subframe, the instantaneous value of jitter, J, is equal to the jitter in the subframe plus the error in estimating f. This error is the time the subframe took to execute minus $\tau(f)$. We can further extend this equation to distribute the jitter across k frames. The equation will be $\tau(f) = \tau(f_0) - J/k$. Henceforth, we refer to k as the *reactivity factor*. The final aim is to set the jitter-count to 0 at the end of the frame. It is difficult to compensate for the jitter in the last few subframes. One solution is to add a few dummy subframes at the end. Our scheme can be trivially extended to model this.

4.2 OS Cache

Destructive interference between the application and the OS in the memory system is a major source of OS jitter. Nellans et al. [14] have suggested the use of an additional cache *OS cache* at the L2 level to segregate the memory accesses of the application and the OS. The accesses of the application and

the OS are sent to their respective caches by checking the value of the supervisor bit. They use an OS cache of the same size as the application cache.

We build on their work. First, we observed that the application epoch is generally bigger than the OS epoch. Hence, we propose the use of a smaller OS cache (64 KB) at the L2 level. Second, we did not obtain appreciable benefits by using their naive approach. This was because there were capacity misses for certain OS epochs, and this nullified the effects of the OS cache. Hence, we propose an intelligent cache in this work that can dynamically share lines between the OS and application caches to effectively mitigate conflict and capacity misses.

The cache lines that are shared between the application and the kernel are stored in the application cache and accesses to such lines are marked using a special *shared bit* in the memory request. Moreover, the TLBs and the page tables are augmented with this extra shared bit such that the request can be sent to the right L2 level cache upon a L1 miss. It is possible to annotate these shared pages by instrumenting the system call handlers in the kernel.

It is often the case that the OS cache is full but there may be some free (invalid) cache lines in the normal L2 cache or vice-versa. We propose a cache line sharing mechanism wherein the application and OS can use some of the space available in the other cache seamlessly. However, in order to restrict the interference due to such sharing, a cache line is treated with least priority when stored in the other cache. We augment the cache logic such that a dedicated bit(overflow bit) in each line of a set is 1 if there is a possibility that a certain line in the set might be present in the other cache. Also, the number of cache lines that can be stored per set in the other cache is limited to half of the associativity of the other cache.

By design, we never store a cache line in both the application and the OS cache. Hence, from the directory's perspective, the combination of the application and the OS caches can be viewed as a single large cache. A detailed description of the OS cache can be found in Appendix C.

5 Evaluation Setup

5.1 Architectural Simulation

Our architectural simulations use the environment shown in Table 2. This is similar to the setup used by [15].

We simulate the Splash2 set of benchmarks [16] using the default inputs for sixteen cores similar to [15]. We had issues in running *cholesky*, *fft*, *volrend* and *radiosity* for the X86-64 architecture in our simulation infrastructure. For *lu* and *ocean*, we use the contiguous_partitions inputs. We simulate the Parsec-2.1 set of benchmarks [17] using native inputs. We had issues in running *canneal*, *blackscholes* and *freqmine* in the Parsec benchmarks suite. This leaves us with eight Splash benchmarks and nine Parsec benchmarks (total 17). In this work, we have chosen general purpose, shared memory, high performance parallel applications, and we have tried to run them in an environment, where we try to dynamically nullify all the jitter. Our main aim has been to ensure scalability and deterministic execution at the level of a frame.

We use an in-house cycle accurate simulator that uses the popular binary instrumenter PIN [18] to simulate the Splash2 [16] and Parsec-2.1 [17] benchmarks. We describe the method of collecting jitter traces and injecting them in our simulations in Appendix D, available in the online supplemental material. We observe that operating system jitter is an inherently random process. Consequently, we repeat the entire process for each benchmark for 10 times, and report the maximum, minimum, and mean time of executions.

Since the benchmarks we considered did not have programmer annotations, we considered the entire program execution to be one single frame. Each subframe is 330 μs long. A subframe should be much larger than the PLL lock time (10 μs) and should be smaller than a OS scheduling quantum(jiffy) (1 millisecond). The first 90 subframes during the simulation of a benchmark are used to warm up the caches and no measurements are taken during this period.

TABLE 2
Simulation Setup

Simulated System Configuration
System : CMP with 16 cores (32 nm process) Core : 2-issue, in-order, 3 GHz(baseline freq) Peak frequency: 3.6 Ghz
L1 : 2-way, 32kB, Private, 32 byte line Hit delay : 2 cycles round-trip, write-back L2 : 4-way, 256kB, Private, 64 byte line, MESI Hit delay : 12 cycles, write-back Miss latency : To other L2s : 30 cycles round-trip(avg) To memory : 300 cycles round-trip OS L2 : 4-way, 64kB, Private, 64 byte line, MESI Hit delay : 3 cycles, write-back Miss latency : To other L2s : 30 cycles round-trip(avg) To memory : 300 cycles round-trip Directory : fully mapped, 32K entries Memory : DDR3 DRAM, dual channel

Properly Tuned System
Ubuntu 9.10 Server, Linux Kernel 2.6.31 (RT patches) Single user mode, RT Priority (80), DVFS turned off SCHED_RR scheduling policy PCI/APIC/Timer interrupts mapped to first two cores Each core handles IPI/machine check polls and sys calls

5.1.1 Area and Power Simulation

We calculate the area and power overheads using Cacti 5.1 [19] and Wattch [20].

Our DVFS settings are given in Table 3. We assume a 10 μs PLL lock time, and a maximum time of 20 μs to ramp up the voltage (see [21]). We assume that our baseline system runs at 3 GHz, which is lower than the rated frequency of 3.6 GHz. All the applications should run optimally in an ideal system running at 3 GHz. Our baseline has a lower frequency than the rated frequency, because we need some additional leeway to increase the frequency to compensate for jitter.

6 Evaluation

6.1 Jitter Characterization

Figs. 7 and 8 show the distribution of the jitter per synchronization operation for the *fmm* (Splash2) and *bodytrack* (Parsec-2.1) benchmarks respectively. Please note that we only plot those values that are above the jitter threshold (10 μs in

TABLE 3
DVFS Factors

Freq (GHz)	V_{dd} (V)	DVFS Factor
2.4	0.8	0.8
2.55	0.85	0.85
2.7	0.9	0.9
2.85	0.95	0.95
3	1	1
3.15	1.05	1.05
3.3	1.1	1.1
3.45	1.15	1.15
3.6	1.2	1.2

our case). We observe a heavy tailed distribution similar to the log-normal distribution. The other benchmarks in the two benchmark suites follow similar distributions. The average jitter is about 100 μs, and starts tapering off after about 200 μs. In some runs, we have observed the jitter to be as high as a couple of milliseconds.

Fig. 9 shows the breakup of the jitter/kernel execution overhead for the simulated benchmarks in Splash2 and Parsec experienced across all the cores. As mentioned above, delays less than the jitter threshold, 10 μs, are not considered as jitter. We show the results for—$u \rightarrow$ unlock, $b \rightarrow$ broadcast, $s \rightarrow$ signal, $b\rightarrow$ broadcast-wait, $lu \rightarrow$ lock-unlock and $sw \rightarrow$ signal-wait jitter (see Fig. 6).

We observe that lock-unlock and just unlock jitter account for a lion's share of the total jitter for some benchmarks. The lock-unlock jitter varies from 2.5 to 81.9 percent. Since prior work [16], [22] has observed that a majority of the synchronization calls are lock operations, we can justify this result. Both the benchmark suites hardly use signal-wait synchronization. The only benchmarks that use it to an appreciable extent are *ferret*, *dedup* and *vips*. Especially, in the case of *vips*, signal-wait jitter is 91.5 percent of all the jitter.

The most important type of jitter is broadcast-wait. The broadcast-wait jitter varies from 12.5 to 91.2 percent. Even though, broadcast calls are relatively rare, its corresponding wait operations are very jitter prone because the waiting thread is typically out of action for a long time. The kernel opportunistically uses this time window to schedule its own tasks. Consequently, there is a visible delay in waking up the waiting task. Second, the library and the kernel also need to wake up several waiting tasks (15 in our case). The last few tasks end up perceiving some jitter. The other interesting result is that with the exception of *x264*, the broadcast jitter is negligible.

Type of jitter	Abbreviation
Signal-Wait	sw
Signal	s
Broadcast-Wait	bw
Broadcast	b
Lock-Unlock	lu
Unlock	u

Fig. 6. Different types of jitter.

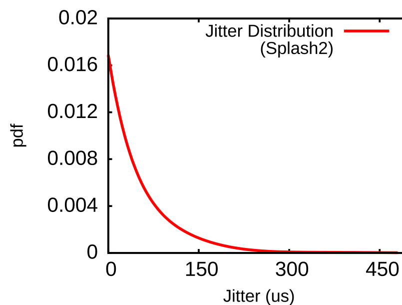

Fig. 7. Jitter (Splash2).

If we compare it with the case of signal, we observe that its proportion is much lower in benchmarks that use it. The signal jitter on an average is about 10-15 percent, whereas the broadcast jitter is about 2-5 percent. This is because, we have a lot of waiting threads in the case of broadcast. The kernel can use their cpu time to do its work. Lastly, as compared to Splash, we see much more diversity in the case of Parsec.

6.1.1 Synch. Jitter versus Total Jitter

In this section, we try to estimate the contribution of synchronization jitter to the total jitter (as defined in Section 2.1). First, we use our jitter traces collected from the actual system by instrumenting the GNU Libc (C standard libraries) to compute the critical path of the program, which can potentially flow across multiple threads. It consists of two parts: (1) pure execution and (2) jitter. We estimated (1) using Linux utilities, and since we know the total time, we can compute (2). Now, the total jitter (2) consists of synchronization jitter, and non-synchronization jitter. Using our jitter measurement infrastructure, we were able to compute the synchronization jitter. Consequently, we were able to get an estimate of the non-synchronization jitter also.

Table 4 plots the ratio of synchronization jitter to total jitter for the Splash benchmarks averaged across all threads. We observe that for six out of the eight benchmarks, the ratio is fairly high. It is between 82 to 97 percent. In fact other than *barnes*, the rest of the Splash benchmarks have figures larger than 95 percent. Without including *lu* and *radix*, the mean is 93 percent. *Lu* and *radix* are kernels. They have very infrequent synchronization operations. Consequently, other sources contribute to most of the jitter.

Table 5 shows the same data for the Parsec benchmarks. Here also the mean value is fairly large, i.e., 91 percent. For benchmarks such as *dedup*, *facesim*, *bodytrack*, and *streamcluster*, the values are larger than 95 percent.

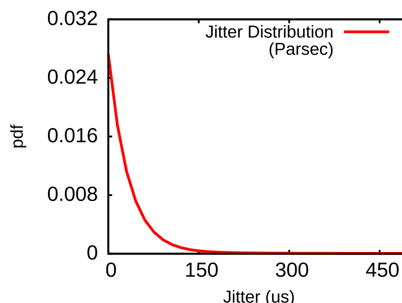

Fig. 8. Jitter (Parsec).

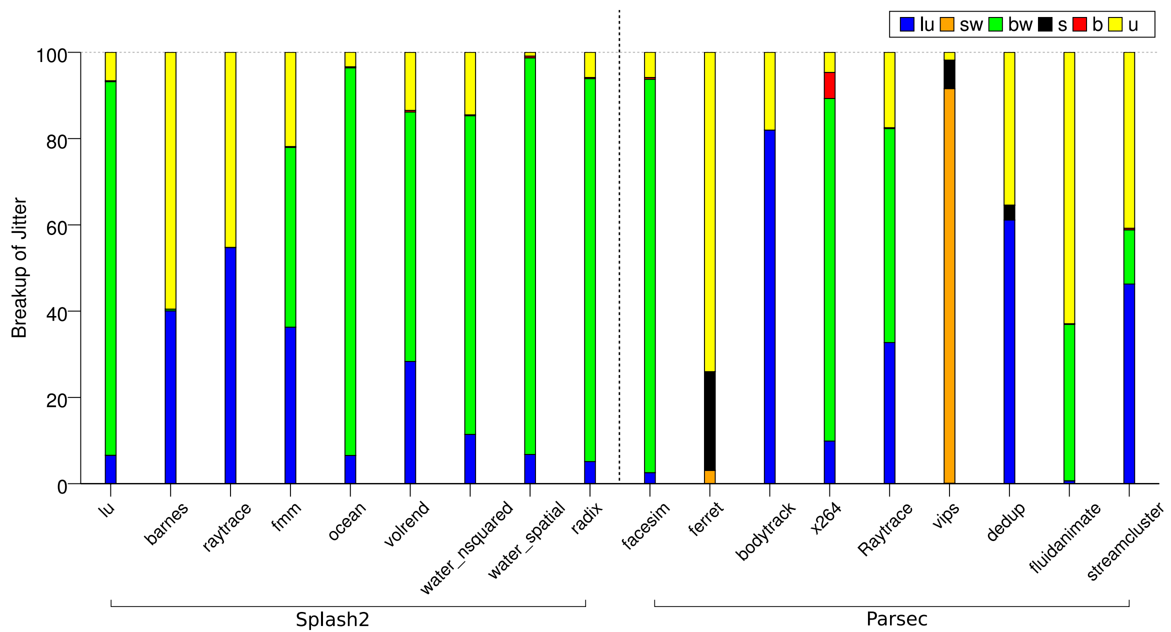

Fig. 9. Break-up of the jitter (Splash2 & Parsec).

6.2 Time Overhead

In this section, we evaluate the time overhead of jitter, and also the efficacy of our proposed scheme. In this section, we discuss the results of two configurations *Unified Cache*, and *OS Cache*. *Unified Cache* models a real system where the application and the kernel share the L2 cache. *OS Cache* is our proposed intelligent OS cache where the application and OS selectively share cache lines to balance interference and performance.

Fig. 10 shows the effects of OS jitter for both the configurations—*Unified Cache* and *OS-Cache*. We run each experiment 10 times (error bars in the figure).

We observe that on an average, 14.5 and 6.4 percent slowdown is experienced by Splash2 and Parsec respectively due to jitter which is significant for high-performance parallel applications.

We observe that using DVFS with inputs from the jitter units on the unified cache, the mean jitter is just 1 percent for Parsec. We also notice that for six out of the nine benchmarks, it is negligible and for the remaining three, *facesim*, *dedup* and *fluidanimate*, it is limited to just 2.5 percent. For Splash, the average jitter is limited to 2.5 percent for five of the eight benchmarks. In case of *water_nsquared*, *water_spatial*, and *ocean*, the jitter could not be mitigated primarily because of power constraints (there is a limit on the amount by which voltage and frequency can be scaled). However, for even these benchmarks, the total jitter falls from the 30-35 percent range to the 10-20 percent range. We also observe that for 13 out of the 17 benchmarks, the variance in the execution times is less than 1 percent.

On the other hand, we notice that, on an average, the OS-Cache performs better than the unified cache (without DVFS). This is expected since the amount of interference is reduced due to the partitioning the application and the kernel accesses. We see significant a benefit by using a separate OS cache in the case of *ferret* and *ocean* where merely using an OS cache mitigates the jitter completely. We attribute this is to: i) the reduction in interference between application and OS (as noticed in *ocean*), and ii) the reduction in the number of capacity misses enabled by flexible sharing of cache lines (as noticed in *ferret*).

Only in 4 out of the 17 benchmarks (*streamcluster*, *facesim*, *raytrace*) and *water_spatial*, the *OS cache* performs worse than the unified cache. This is because there were too many capacity misses that the OS cache could not accommodate. Even in these benchmarks, the performance of *OS cache* is close to that (<2 percent) of the unified cache configuration.

However, the most pernicious aspect of jitter is the non-determinism in execution times for the same benchmark across multiple runs. Let us consider some examples. In the Splash benchmark suite, the total execution time of *water_nsquared* and *water_spatial* vary up to 11 and 17 percent respectively. We observe that not only OS jitter

TABLE 4
Proportion of Synch Jitter (Splash2)

app	$\frac{synch.jitter}{totjitter}$	app	$\frac{synch.jitter}{totjitter}$
barnes	82%	*fmm*	95%
ocean	97%	*raytrace*	97%
water-nsq	91%	*lu*	30%
radix	33%	*water-sp*	95%
Mean (6 out of 8)	93%		

TABLE 5
Proportion of Synch Jitter (Parsec)

app	$\frac{synch.jitter}{totjitter}$	app	$\frac{synch.jitter}{totjitter}$
facesim	98%	*ferret*	80%
bodytrack	96%	*x264*	86%
raytrace	83%	*vips*	88%
dedup	97%	*fluidanimate*	96%
streamcluster	99%	Mean	91%

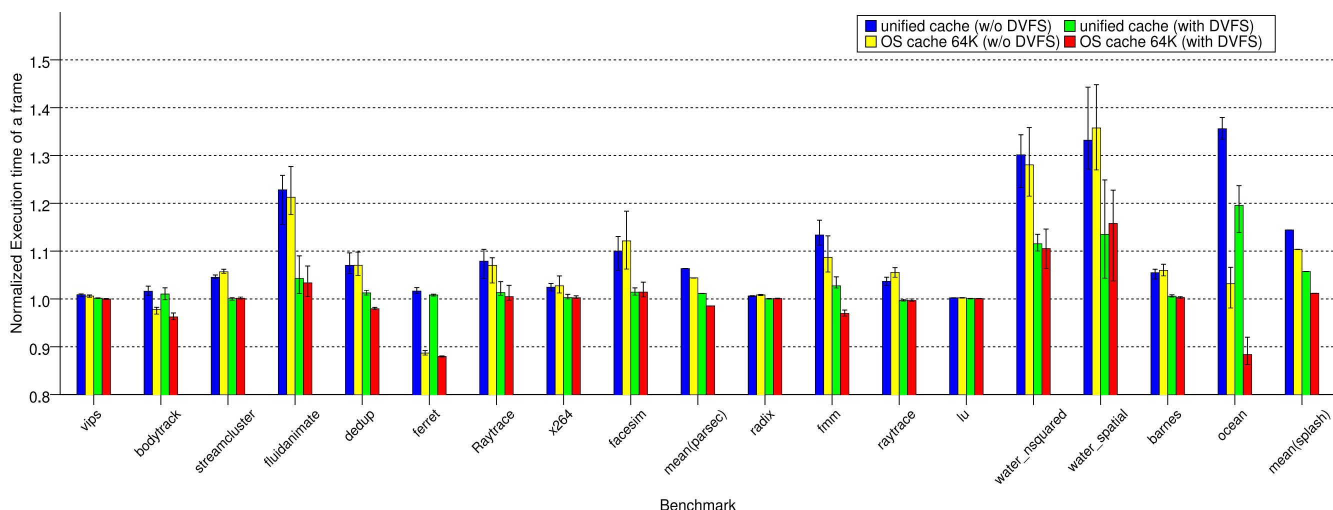

Fig. 10. Time overhead (Splash2 & Parsec).

leads to a net slowdown, it also introduces a substantial variance in the execution time. This makes it difficult to design high performance parallel applications.

We observe that a combination of intelligent DVFS and having the OS cache completely mitigates jitter. For Parsec, DVFS combined with OS cache gives a speedup of 1.5 percent whereas for Splash, the overall mean jitter is just 1 percent. The observed speedups are due to the controller over-compensating for the observed jitter in some cases. We also observe that in only 3 out of 17 benchmarks, jitter has not been fully mitigated. As previously mentioned, this is due to the limit on the amount of voltage-frequency scaling that is possible on a given system.

6.3 Power Overhead

In this section, we evaluate the power overhead of our scheme. First, we observe that since the jitter unit is only used when we have a synchronization event (typically once every 50 μs), or at the beginning of a subframe, the power overhead of the jitter unit per se is negligible.

Figs. 11 and 12 show the normalized frequency settings for a typical frame across 30 subframes for *water_spatial* and *facesim* respectively. We observe the responsiveness of our frequency scaling algorithm. When the jitter has been overcompensated in one subframe, we notice that the voltage-frequency scaling algorithm tries to minimize the power overhead by dropping the supply voltage to a value lower than the base voltage (seen in the dip of frequency to 0.95). We further observe that in the case of *water_spatial*, the amount of jitter is high, and consequently the controller tries its best to control it by setting the highest possible DVFS factor, 1.2. However, when the value of the jitter drops, the controller realizes that it has over compensated, and then tries to save power.

Fig. 13 shows the normalized power overhead. The power overhead varies from less than 1 to 41 percent. For the Splash benchmarks, the average power overhead is 14 and 13.6 percent for *Unified* and *OS-Cache* configurations respectively. Whereas, for Parsec, the corresponding numbers are 16.3 and 16.5 percent respectively. For 11 out of 17 benchmarks, the average overhead is limited to 20 percent in both the configurations.

As expected the values of power consumption are roughly correlated with the values of measured jitter shown in Fig. 10. For example, benchmarks such as *lu*, *ferret*, and *x264*, have low values of jitter and power. *Ocean*, *facesim*, *fluidanimate*, and *fmm* have high values of jitter, and high power consumption also.

However, there are some exceptions such as *bodytrack*. They have low values of jitter, and still have high power consumption. Likewise, we have benchmarks such as *water_nsquared*, and *water_spatial*, which show the reverse trend. After studying these benchmarks, we could explain this phenomenon on the basis of the nature of the critical

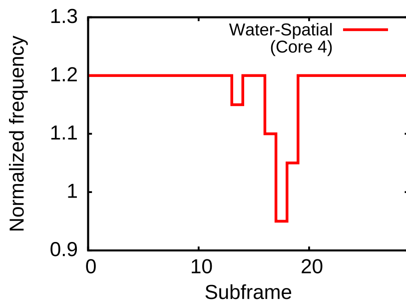

Fig. 11. Frequency across subframes (*water_spatial*).

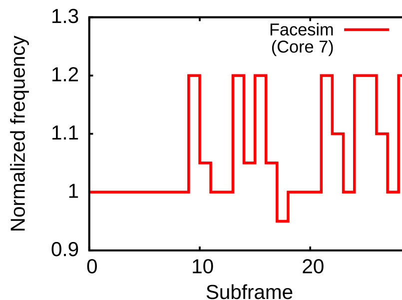

Fig. 12. Frequency across subframes (*facesim*).

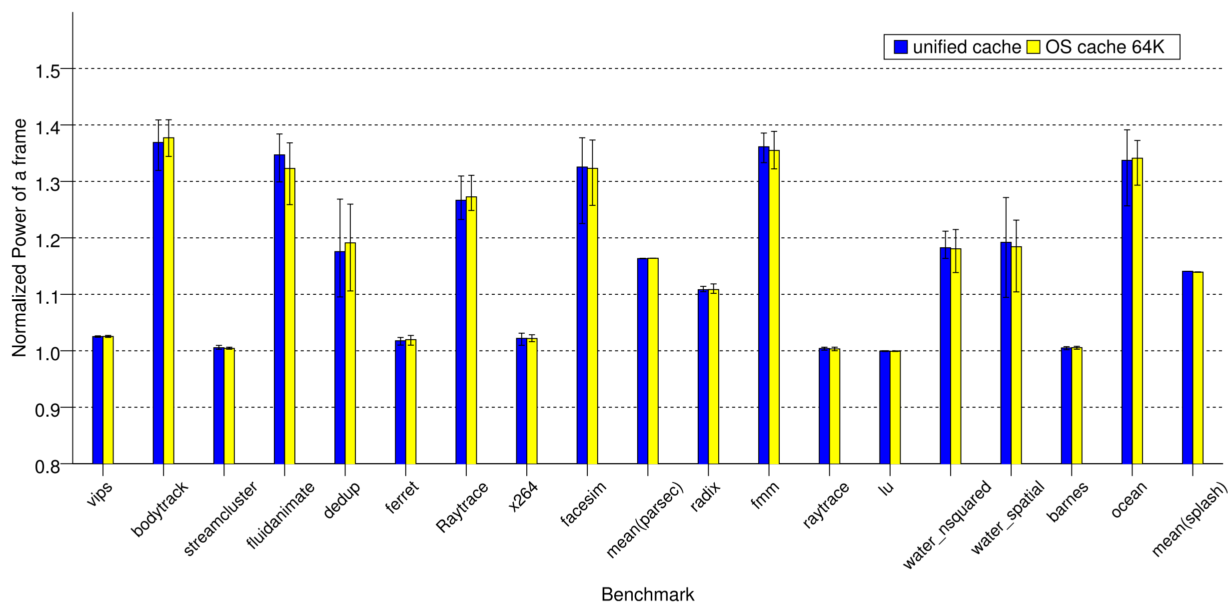

Fig. 13. Power overhead (Splash2 & Parsec).

path. In some benchmarks all the jitter happens on the critical path. Consequently, the power overhead is relatively low. As compared to this, in some other benchmarks there is a lot of jitter that happens in executions that are off the critical path. Since, the controller does not have instantaneous knowledge of the critical path, it needs to nullify jitter locally and synchronize later (see Section 4.1.4). This leads to higher power consumption.

6.4 Area Overhead

We synthesized the jitter unit described in Section 4 using the UMC 90 nm technology standard cell library. The implementation of the jitter unit uses a wait buffer and lock buffer of eight entries each. We do not observe any space overflows in our simulations. The synthesized jitter unit occupies 46,166 $\mu\mathrm{m}^2$ and has a delay of 850 ps.

Using standard technology scaling rules [23], we project the size of the jitter unit to be 8,550 $\mu\mathrm{m}^2$ per core for a 32 nm process. On a chip with 16-cores, the total area occupied by jitter units is 0.136 mm^2. The size of the (64 kB) OS cache obtained from Cacti 5.1 [19] is 0.975 mm^2, and for all the 16 cores, it occupies a total area of 3.9 mm^2. Assuming a 400 mm^2 die, the total area overhead is: 0.034 percent (jitter units) + 0.97 percent (OS cache). Therefore, our proposed method with the OS cache and jitter units has an area overhead of approximately 1 percent, which is small.

7 Conclusion

In this paper, we proposed a scheme to measure, characterize, and mitigate the effects of operating system jitter on CMP based parallel programs. We proposed to have intelligent performance counters called jitter units on every core. These jitter units record thread synchronization events, which are generated by an instrumented version of the C library along with bus events, context switches, and power management events. Second, we proposed an adaptive algorithm which distributes the compensation of jitter over next few subframes based on the interference due to OS at the L2 cache. This scheme was not sufficient to completely nullify jitter. Hence, we augmented the design with an OS cache that saves the cache lines belonging to the kernel, modules, and drivers. It can intelligently trade lines between the itself and the regular application cache.

We showed in Section 6.1.1 that the main contributor to OS jitter in CMP based parallel programs is thread synchronization events. Subsequently, we characterized the sources of synchronization jitter and found broadcast-wait methods to be the largest contributor. We showed in Section 6 that we are able to decrease the total amount of jitter from 14.5 to 1 percent for the Splash2 benchmark suite and from 6.4 to 0 percent for the Parsec Benchmark Suite. We can almost completely nullify jitter for 12 of the 17 benchmarks. Our scheme has a mean power overhead of approximately 15 percent for all the simulated benchmarks. Lastly, we evaluated the area overheads of our scheme, and found it to be approximately 1 percent.

References

[1] M. Lee, Y. Ryu, S. Hong, and C. Lee, “Performance Impact of Resource Conflicts on Chip Multi-Processor Servers,” *Proc. Eighth Int’l Conf. Applied Parallel Computing: State of the Art in Scientific Computing (PARA ’06)*, pp. 1168-1177, 2007.

[2] R. Gioiosa, S. McKee, and M. Valero, “Designing OS for HPC Applications: Scheduling,” *Proc. IEEE Int’l Conf. Cluster Computing (CLUSTER)*, pp. 78-87, Sept. 2010.

[3] P. De, V. Mann, and U. Mittal, “Handling OS Jitter on Multicore Multithreaded Systems,” *Proc. IEEE Int’l Symp. Parallel and Distributed Processing (IPDPS)*, 2009.

[4] F. Petrini, D.J. Kerbyson, and S. Pakin, “The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8, 192 Processors of ASCI Q,” *Proc. ACM/IEEE Conf. High Performance Networking and Computing*, 2003.

[5] T. Jones, S. Dawson, R. Neely, W. Tuel, L. Brenner, J. Fier, R. Blackmore, P. Caffrey, B. Maskell, P. Tomlinson, and M. Roberts, “Improving the Scalability of Parallel Jobs by Adding Parallel Awareness to the Operating System,” *Proc. ACM/IEEE Conf. Supercomputing (SC ’03)*, p. 10, http://doi.acm.org/10.1145/1048935.1050161, 2003.

[6] P. Terry, A. Shan, and P. Huttunen, "Improving Application Performance on HPC Systems with Process Synchronization," *Linux J.*, vol. 2004, no. 127, p. 3, 2004.
[7] P. De, R. Kothari, and V. Mann, "Identifying Sources of Operating System Jitter through Fine-Grained Kernel Instrumentation," *Proc. IEEE Int'l Conf. Cluster Computing*, pp. 331-340, 2007.
[8] M. Chetlur, U. Devi, P. Dutta, P. Gupta, L. Chen, Z.B. Zhu, S. Kalyanaraman, and Y. Lin, "A Software Wimax Medium Access Control Layer Using Massively Multithreaded Processors," *IBM J. Research and Development*, vol. 54, no. 1, pp. 1–13, 2010.
[9] L. Nuaymi, *WiMAX: Technology for Broadband Wireless Access*. John Wiley & Sons, 2007.
[10] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole, "A Measurement-Based Analysis of the Real-Time Performance of Linux," *Proc. Eighth IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, 2002.
[11] M. Paolieri, N.E. Qui, F.J. Cazorla, G. Bernat, and M. Valero, "Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems," *Proc. 36th Ann. Int'l Symp. Computer Architecture (ISCA)*, 2009.
[12] S.R. Sarangi, B. Greskamp, A. Tiwari, and J. Torrellas, "EVAL: Utilizing Processors with Variation-Induced Timing Errors," *Proc. 41st IEEE/ACM Int'l Symp. Microarchitecture (MICRO)*, 2008.
[13] T. Sherwood, S. Sair, and B. Calder, "Phase Tracking Prediction," *Proc. 30th Int'l Symp. Computer Architecture (ISCA)*, 2003.
[14] D. Nellans, R. Balasubramonian, and E. Brunvand, "Interference Aware Cache Designs for Operating System Execution," Technical Report UUCS-09-002, Univ. of Utah, Feb. 2009.
[15] R. Agarwal and J. Torrellas, "Flexbulk: Intelligently Forming Atomic Blocks in Blocked-Execution Multiprocessors to Minimize Squashes," *Proc. 38th Ann. Int'l Symp. Computer Architecture (ISCA)*, 2011.
[16] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *ACM SIGARCH Computer Architecture News*, vol. 23, pp. 24-36, May 1995.
[17] C. Bienia, "Benchmarking Modern Multiprocessors," PhD dissertation, Princeton Univ., Jan. 2011.
[18] V.R.A. Settle, D. Connors, and R. Cohn, "PIN: A Binary Instrumentation Tool for Computer Architecture Research Education," *Proc. Workshop on Computer Architecture Education (WCAE)*, 2004.
[19] S. Thoziyoor, N. Muralimanohar, J. Ahn, and N.P. Jouppi, "CACTI 5.1," Technical Report HPL-2008-20, HP Labs, 2008.
[20] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A Framework for Architectural-Level Power Analysis and Optimizations," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 83-94, May 2000.
[21] J. Suh and M. Dubois, "Dynamic MIPS Rate Stabilization in Out-of-Order Processors," *Proc. 36th Ann. Int'l Symp. Computer Architecture (ISCA)*, 2009.
[22] C. Bienia, S. Kumar, J.P. Singh, and K. Li, "The Parsec Benchmark Suite: Characterization and Architectural Implications," *Proc. 17th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT '08)*, pp. 72-81, 2008.
[23] W. Huang, K. Rajamani, M. Stan, and K. Skadron, "Scaling with Design Constraints: Predicting the Future of Big Chips," *IEEE Micro*, vol. 31, no. 4, pp. 16-29, July/Aug. 2011.
[24] D. Tsafrir, Y. Etsion, D.G. Feitelson, and S. Kirkpatrick, "System Noise OS Clock Ticks Fine-Grained Parallel Applications," *Proc. 19th Ann. Int'l Conf. Supercomputing (ICS)*, 2005.
[25] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole, "Supporting Time-Sensitive Applications on a Commodity OS," *ACM SIGOPS Operating System Rev.*, vol. 36, no. SI, pp. 165-180, 2002.
[26] K.B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing Application Sensitivity to OS Interference Using Kernel-Level Noise Injection," *Proc. ACM/IEEE Conf. Supercomputing (SC)*, 2008.
[27] D. Tsafrir, "The Context-Switch Overhead Inflicted by Hardware Interrupts (and the Enigma of Do-Nothing Loops)," *Proc. Experimental Computer Science*, 2007.
[28] R. Love, *Linux Kernel Development*. Addison-Wesley, 2010.
[29] F. Hubertus and R. Rusty, "Fuss Futexes Furwocks: Fast Userlevel Locking in Linux," *Proc. Ottawa Linux Symp.*, 2002.
[30] "Linux Kernel Archives," git://git.kernel.org/pub/scm/linux/kernel/git/maxk/cpuisol-2.6.git, 2012.
[31] S. Baskiyar and N. Meghanathan, "A Survey of Contemporary Real-Time Operating Systems," *Informatica (Slovenia)*, vol. 29, no. 2, pp. 233-240, 2005.
[32] F. Bellard, "QEMU, A Fast Portable Dynamic Translator," *Proc. Ann. Conf. USENIX Ann. Technical Conf.*, 2005.
[33] P.S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A Full System Simulation Platform," *Computer*, vol. 35, no. 2, pp. 50-58, 2002.
[34] P. De and V. Mann, "jitSim: A Simulator for Predicting Scalability of Parallel Applications in Presence OS Jitter," *Proc. 16th Int'l Euro-Par Conf. Parallel Processing (Europar)*, 2010.
[35] D. Freedman, R. Pisani, and R. Purves, *Statistics*. W.W. Norton and Company, 2007.

Sandeep Chandran received the bachelor's degree in computer science and engineering from Visveswaraya Technological University. He is currently a research scholar at the Department of Computer Science & Engineering, Indian Institute of Technology, Delhi. Prior to joining the PhD program, he has worked in the industry for two years. His research interests include post-silicon validation methodologies, architectural design-space exploration and fault-tolerant systems.

Prathmesh Kallurkar received the bachelor's degree in computer science and engineering from Birla Vishvakarma Mahavidyalaya, Sardar Patel University, and the master's degree in computer science from the Department of Computer Science and Engineering, Indian Institute of Technology, Delhi, where he is currently a research scholar. His research interests include architectural support for operating Systems, and fault-tolerant systems.

Parul Gupta received the BTech degree in electrical engineering from the Indian Institute of Technology, Bombay, and the MS degree in electrical engineering from the University of California, Los Angeles. She is currently a technical staff member with IBM Research-India. Her research interests include algorithms for wireless communication systems, cloud computing, green technologies and analytics. She has co-authored 12 publications and six patents. She is a senior member of the IEEE and the ACM.

Smruti R. Sarangi received the BTech degree in computer science from IIT Kharagpur, India, in 2002, and the MS and PhD degrees in computer architecture from the University of Illinois at Urbana-Champaign in 2007. He is currently an assistant professor in the Department of Computer Science and Engineering, IIT Delhi, India. He has spent four years in industry working in IBM India Research Labs, and Synopsys. He works in the areas of computer architecture, parallel and distributed systems. He is a member of the IEEE and the ACM.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.