# Architectural Support for Handling Jitter in Shared Memory based Parallel Applications

Sandeep Chandran, Prathmesh Kallurkar, Parul Gupta, and Smruti R. Sarangi,

**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 (upto 35.22%) 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% for 14 out of 17 benchmarks with modest DVFS factors. We reduce the overall jitter by an average 13.5% for Splash2 and 6.4% for Parsec. The area overhead of our scheme is limited to 1%.

**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 HPC(High Performance Computing) 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 upto 41% and 27% (see Figure 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% 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/seconds 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

- *Sandeep Chandran, Prathmesh Kallurkar and Smruti R. Sarangi are with the Department of Computer Science & Engineering, Indian Institute of Technology Delhi, New Delhi – 110016*
- *Parul Gupta is with IBM Research Labs, India.*

cues from special instructions inserted into the standard POSIX thread library and the 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$\mu$s 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. 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 (upto 85%). 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.

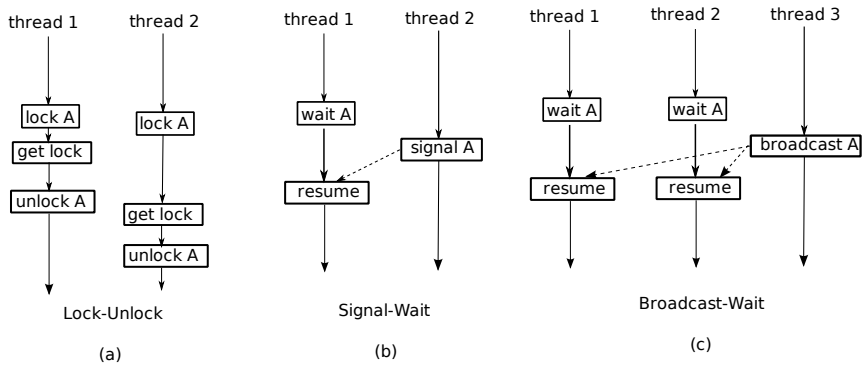
Passive jitter is caused as a side effect of system calls. In parallel benchmarks, synchronization calls (mutex lock, unlock, etc.) often lead to system calls. The kernel uses these opportunities to schedule its own work to run book keeping tasks, daemons, or bottom-halves of device drivers. In our studies 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

Figure 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.




| 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 |

TABLE 1:
List of events

Fig. 1: Synchronization primitives in the pthread library

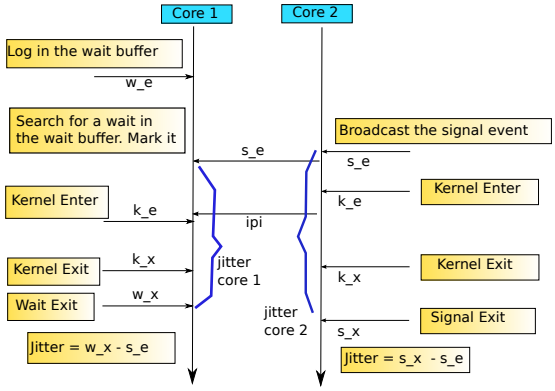



Fig. 2: Synchronization involving two cores

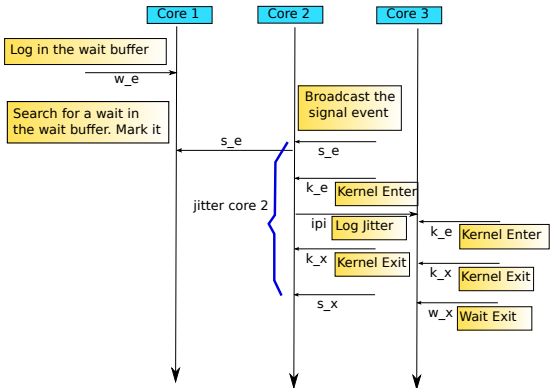



Fig. 3: Synchronization involving three cores

### 3.2 Scenarios for Signal-Wait Synchronization

#### *3.2.1 Case 1*

In Figure 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. 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 0x80 instruction on x86). The kernel immediately runs the schedule function, in which the kernel can either send an inter-processor-interrupt to core 1 to resume the waiting thread, or it can mark the waiting thread as ready, and just return. Figure 2 shows the former case. The latter case is treated the same way.

The time between s_e and ipi (inter processor interrupt) 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 Figure 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 $N\mu s$ 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.

#### 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).

#### 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 Figure 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).

We consider these cases **exhaustive** since we observe in our experiments that their coverage is more than 99.999%. 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 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 Figure 15. 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 (program counter) 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 program counter 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.

**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. We summarize the main structures in this section. The high level design is shown in Figure 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) program counters 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 PC (program counter) 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.

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

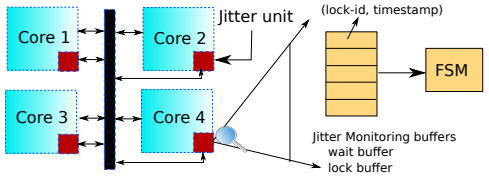



Fig. 4: High-level design of the jitter unit (One per core)

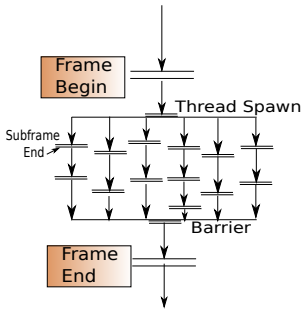



Fig. 5: Subframes in a multi-threaded code

| 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

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 Figures 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.

#### 4.1.4 Calculating the Critical Path

Figure 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]:

$$P = \frac{f}{CPI_{comp} + mr * mp} \quad (1)$$

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. Firstly, 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. Secondly, 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 Section C.1.

## 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 8 Splash benchmarks and 9 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. 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$\mu$s long. A subframe should be much larger than the PLL lock time (10$\mu$s) and should be smaller than a OS scheduling quantum(jiffy) (1 milli-second). 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.

#### 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$\mu s$ PLL lock time, and a maximum time of 20 $\mu s$ 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.

| Simulated System Configuration |
|---|
| System : CMP with 16 cores (32 nm process)<br>Core : 2-issue, in-order, 3 GHz(baseline freq)<br>Peak frequency: 3.6 Ghz |
| L1 : 2-way, 32kB, Private, 32 byte line<br>Hit delay : 2 cycles round-trip, write-back<br>L2 : 4-way, 256kB, Private, 64 byte line, MESI<br>Hit delay : 12 cycles, write-back<br>Miss latency :<br>To other L2s : 30 cycles round-trip(avg)<br>To memory : 300 cycles round-trip<br>OS L2 : 4-way, 64kB, Private, 64 byte line, MESI<br>Hit delay : 3 cycles, write-back<br>Miss latency :<br>To other L2s : 30 cycles round-trip(avg)<br>To memory : 300 cycles round-trip<br>Directory : fully mapped, 32K entries<br>Memory : DDR3 DRAM, dual channel |
| Properly Tuned System |
| Ubuntu 9.10 Server, Linux Kernel 2.6.31 (RT patches)<br>Single user mode, RT Priority (80), DVFS turned off<br>SCHED_RR scheduling policy<br>PCI/APIC/Timer interrupts mapped to first two cores<br>Each core handles IPI/machine check polls and sys calls |

TABLE 2: Simulation Setup

| 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 |

TABLE 3: DVFS factors

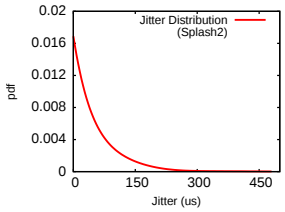



Fig. 7: Jitter (Splash2)

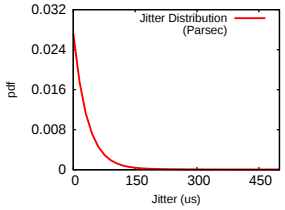



Fig. 8: Jitter (Parsec)

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% |

# 6 Evaluation

## 6.1 Jitter Characterization

Figures 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 $\mu s$ in 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 $\mu s$, and starts tapering off after about 200 $\mu$s. In some runs, we have observed the jitter to be as high as a couple of milli-seconds.

Figure 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 $\mu$s, are not considered as jitter. We show the results for – $u \rightarrow$ unlock , $b \rightarrow$ broadcast, $s \rightarrow$ signal, $bw \rightarrow$ broadcast-wait, $lu \rightarrow$ lock-unlock and $sw \rightarrow$ signal-wait jitter (see Figure 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%. Since prior work [22], [16] 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% of all the jitter.

The most important type of jitter is broadcast-wait. The broadcast-wait jitter varies from 12.5 to 91.2%. 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. Secondly, 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.

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%, whereas the broadcast jitter is about 2-5%. 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

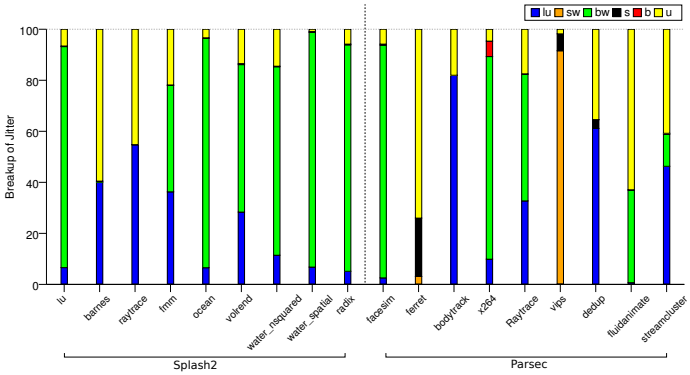



Fig. 9: Break-up of the jitter (Splash2 & Parsec)

diversity in the case of Parsec.

#### 6.1.1 Synch. Jitter vs 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 6 out of the 8 benchmarks, the ratio is fairly high. It is between 82 to 97%. In fact other than *barnes*, the rest of the Splash benchmarks have figures larger than 95%. Without including *lu* and *radix*, the mean is 93%. *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%. For benchmarks such as *dedup*, *facesim*, *bodytrack*, and *streamcluster*, the values are larger than 95%.

### 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.

Figure 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% 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% for Parsec. We also notice that for 6 out of the 9 benchmarks, it is negligible and for the remaining 3, *facesim*, *dedup* and *fluidanimate*, it is limited to just 2.5%. For Splash, the average jitter is limited to 2.5% for 5 of the 8 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% range to the 10-20% range. We also observe that for 13 out of the 17 benchmarks, the variance in the execution times is less than 1%.

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 ($< 2\%$) 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 upto 11% and 17% respectively. We observe that not only OS jitter

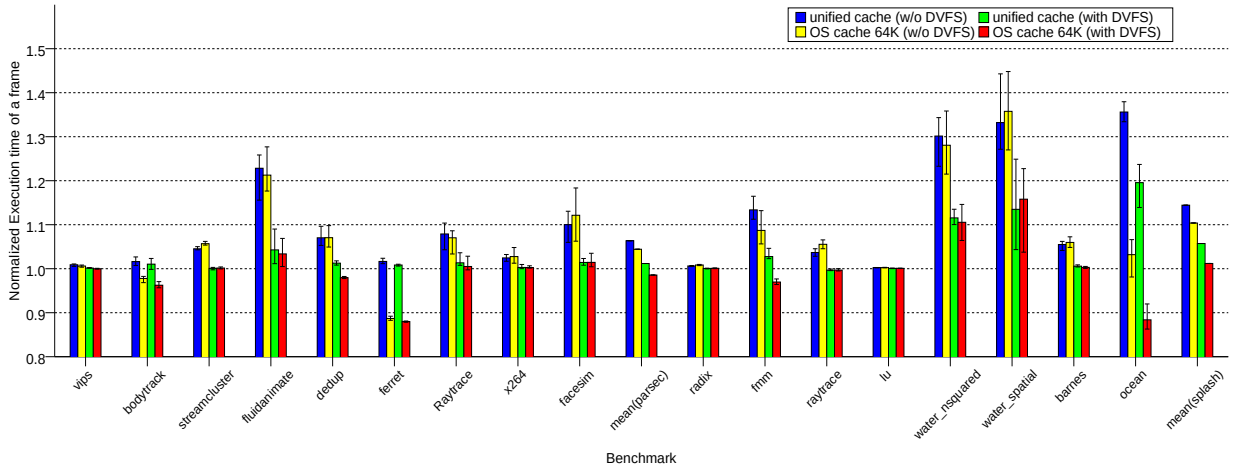



Fig. 10: Time overhead (Splash2 & Parsec)

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% whereas for Splash, the overall mean jitter is just 1%. 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$\mu s$), or at the beginning of a subframe, the power overhead of the jitter unit per se is negligible.

Figures 11 and 12 shows 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.

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

As expected the values of power consumption are roughly correlated with the values of measured jitter shown in Figure 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 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 90nm technology standard cell library. The implementation of the jitter unit uses a wait buffer and lock buffer of 8 entries each. We do not observe any space overflows in our simulations. The synthesized jitter unit occupies 46166 $\mu m^2$ and has a delay of 850 $ps$.

Using standard technology scaling rules [23], we project the size of the jitter unit to be 8550 $\mu m^2$ 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

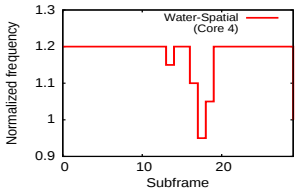



Fig. 11: Frequency across subframes (*water_spatial*)

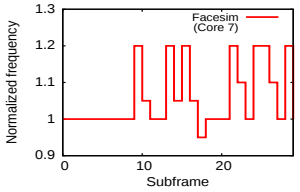



Fig. 12: Frequency across subframes (*facesim*)

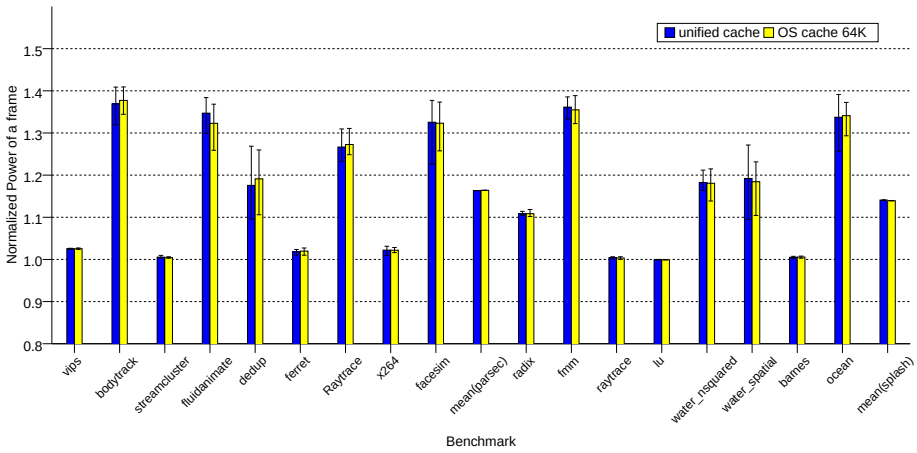



Fig. 13: Power overhead (Splash2 & Parsec)

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%(jitter units) + 0.97% (OS cache). Therefore, our proposed method with the OS cache and jitter units has an area overhead of approximately 1%, 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. Secondly, 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% for the Splash2 benchmark suite and from 6.4% to 0% 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% for all the simulated benchmarks. Lastly, we evaluated the area overheads of our scheme, and found it to be approximately 1%.

## References

[1] M. Lee, Y. Ryu, S. Hong, and C. Lee, "Performance impact of resource conflicts on chip multi-processor servers," in *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, ser. PARA'06, 2007, pp. 1168–1177.

[2] R. Gioiosa, S. McKee, and M. Valero, "Designing os for hpc applications: Scheduling," in *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, sept. 2010, pp. 78 –87.

[3] P. De, V. Mann, and U. Mittal, "Handling os jitter on multicore multithreaded systems," in *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," in *SC*, 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," in *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, ser. SC '03. New York, NY, USA: ACM, 2003, pp. 10–. [Online]. Available: http://doi.acm.org/10.1145/1048935.1050161

[6] P. Terry, A. Shan, and P. Huttunen, "Improving application performance on hpc systems with process synchronization," *Linux Journal*, vol. 2004, no. 127, 2004.

[7] P. De, R. Kothari, and V. Mann, "Identifying sources of operating system jitter through fine-grained kernel instrumentation," in *Cluster*, 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 Journal of Research and Development*, vol. 54, no. 1, 2010.

[9] L. Nuaymi, *WiMAX: Technology for Broadband Wireless Access*. Wiley Publishing, 2007.

[10] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole, "A measurement-based analysis of the real-time performance of linux," in *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," in *ISCA*, 2009.

[12] S. R. Sarangi, B. Greskamp, A. Tiwari, and J. Torrellas, "Eval: Utilizing processors with variation-induced timing errors," in *MICRO*, 2008.

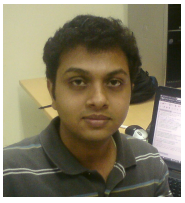
[13] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *ISCA*, 2003.

[14] D. Nellans, R. Balasubramonian, and E. Brunvand, "Interference aware cache designs for operating system execution," University of Utah, Tech. Rep. UUCS-09-002, February 2009.

[15] R. Agarwal and J. Torrellas, "Flexbulk: Intelligently forming atomic blocks in blocked-execution multiprocessors to minimize squashes," in *ISCA*, 2011.

[16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: characterization and methodological consid-

erations," *SIGARCH Comput. Archit. News*, vol. 23, pp. 24–36, May 1995.
[17] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.
[18] V. R. A. Settle, D. Connors, and R. Cohn, "Pin: A binary instrumentation tool for computer architecture research and education," in *WCAE*, 2004.
[19] S. Thoziyoor, N. Muralimanohar, J. Ahn, and N. P. Jouppi, "Cacti 5.1," Tech. Rep. HPL-2008-20, 2008.
[20] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: a framework for architectural-level power analysis and optimizations," *SIGARCH Comput. Archit. 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," in *ISCA*, 2009.
[22] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, ser. PACT '08. New York, NY, USA: ACM, 2008, pp. 72–81.
[23] W. Huang, K. Rajamani, M. Stan, and K. Skadron, "Scaling with design constraints: Predicting the future of big chips," *Micro, IEEE*, 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, and fine-grained parallel applications," in *ICS*, 2005.
[25] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole, "Supporting time-sensitive applications on a commodity os," *SIGOPS Oper. Syst. 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," in *SC*, 2008.
[27] D. Tsafrir, "The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops)," in *Experimental Computer Science*, 2007.
[28] R. Love, *Linux Kernel Development*. Addison-Wesley, 2010.
[29] F. Hubertus and R. Rusty, "Fuss, futexes and furwocks: Fast userlevel locking in linux," in *Ottawa Linux Symposium*, 2002.
[30] "Linux kernel archives," git://git.kernel.org/pub/scm/linux/kernel/git/maxk/cpuisol-2.6.git.
[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 and portable dynamic translator," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, 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," in *Europar*, 2010.
[35] D. Freedman, R. Pisani, and R. Purves, *Statistics*. W. W. Norton and Company, 2007.



**Sandeep Chandran** is a research scholar at the Department of Computer Science & Engg, Indian Institute of Technology, Delhi. He has a Bachelors' degree from Visveswaraya Technological University in Computer Science & Engg. Prior to joining the Ph.D program, he has worked in the industry for 2 years. His research interests include post-silicon validation methodologies, architectural design-space exploration and fault-tolerant systems.

**Prathmesh Kallurkar** is a research scholar at the Department of Computer Science & Engg, Indian Institute of Technology, Delhi. He completed his Master's degree in Computer Science from Department of Computer Science & Engg, Indian Institute of Technology, Delhi. He has a Bachelors' degree in Computer Science & Engg from Birla Vishvakarma Mahavidyalaya, Sardar Patel university. His research interests include architectural support for operating Systems, and fault-tolerant systems.

**Parul Gupta** has a B.Tech.in electrical engineering from the Indian Institute of Technology, Bombay, and M.S. in electrical engineering from University of California, Los Angeles. She is currently a Technical Staff Member with IBM Research - India. Her research interests span algorithms for wireless communication systems, cloud computing, green technologies and analytics. She is a senior member of IEEE and ACM, and has co-authored 12 publications and 6 patents.

**Smruti R. Sarangi** is 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 graduated with a M.S and Ph.D in computer architecture from the University of Illinois at Urbana-Champaign in 2007, and a B.Tech in computer science from IIT Kharagpur, India, in 2002. He works in the areas of computer architecture, parallel and distributed systems. Prof. Sarangi is a member of the IEEE and ACM.

# APPENDIX A
# SOURCES OF JITTER

## A.1 OS Jitter

### A.1.1 Causes

OS jitter has been studied heavily in [3], [7], [4], [5], [6], [24], [25], [26], [27]. When a single application is running, which is typically the case for parallel applications, the main source of jitter is kernel activity in the form of daemons, periodic tasks, and interrupt handlers. At any point of time, the kernel has a work queue of pending work items. Based on relative priorities, the kernel decides to schedule either threads of the user application or threads to do its own work.

There are two types here – active and passive. In the *active* case, the kernel programs the timer chip to deliver periodic timer interrupts (typically once every 10ms). After receiving a timer interrupt, the kernel schedules some of its own threads, and then the user process resumes. In the *passive* case, the user process invokes the kernel through system calls (I/O, network, synchronization). The kernel steals such opportunities to do some of its own work. Consequently, system calls appear to take longer than usual.

**Active** According to several studies [3], [7], [24] (conducted for large MPI programs), the timer interrupt is the single largest cause of OS jitter. It accounts for around 85% of the OS jitter [3]. According to these studies, timer interrupt processing can take upto 3-4 ms. The high contribution of timer interrupts to jitter is not because processing timer interrupts takes a long time and their servicing overhead has higher algorithmic complexity [3]. It is because the timer interrupt is typically the most frequent interrupt, and hence the kernel opportunistically uses it to schedule its own tasks. Other interrupts that are I/O and network related have typically much higher processing overheads. However, in modern processors and operating systems a lot of those latencies are hidden by using DMA, user space I/O, and double buffering [28]. Moreover, the Linux kernel very quickly services an interrupt handler(top half) and returns. Most of the real interrupt processing work is spawned off as a separate thread called the bottom half [3], [7], which is scheduled later. As observed by Mann et. al. [3], [7], the bottom halves are typically scheduled when a timer interrupt is processed.

**Passive** We observe that the main sources of jitter in parallel POSIX thread based applications are synchronization calls (see Table 4 and 5), and not timer interrupts. Table 4 and 5 further show that the jitter due to synchronization operations is 93% and 91% of the total jitter for the Splash2 and Parsec benchmarks respectively. The reason for this is that the number of interrupts to the kernel (sleep/wakeup threads) caused by lock, signal and barrier pthread operations greatly outnumber the number of timer and I/O interrupts. Please note that in the latest implementation of pthreads [29], a synchronization call such as a lock or barrier operation is divided into two stages. In the first stage, the library enters a spin lock to test if the operation can proceed. After a couple of micro-seconds, if the thread hasn't been successful, then it makes a system call, and gets swapped out. We observe that such system calls are fairly frequent, at least as compared to other sources of interrupts. Such calls give the kernel more opportunities to schedule its work items when it is invoked because of these interrupts.

### A.1.2 Solutions

Here is a list of solutions proposed in prior work.

**CPU Isolation:** The kernel schedules all of its work on a single set of cpus. There is an experimental patch to the Linux kernel that implements it and can be obtained from [30]. This scheme [3], [24], [4] is typically very effective in reducing active jitter for MPI based programs. A variant of this approach is a part of our baseline system.

**Real Time Operating Systems:** Real time operating systems like RT Linux, QNX, and VXworks, make all parts of the kernel including interrupt handlers pre-emptible. They are able to make timing guarantees for soft real time systems because user processes have kernel level priorities. The issues with such platforms are that they are typically proprietary, or like real time Linux are tied to specific hardware [31]. This precludes us from running a host of commercial and open source applications [31]. According to Abeni et. al. [10] variants of real time Linux are not suitable for running high performance applications in user space primarily because such kernels have poor performance, and very poor interrupt response characteristics. They observe that the resultant slowdown can be upto 20%.

**Jitter Synchronization:** For tree based communication patterns like barriers, it is possible for jitter across the nodes to add up. This can have very serious consequences. Hence, there are techniques [5], [6] to stop this from happening by co-ordinating the jitter introducing threads across nodes. However, this scheme has limited efficacy, and does not bode well with POSIX thread based parallelism.

## A.2 Jitter due to Multiple Threads

When we have multiple threads, they compete for shared resources like cache lines, memory frames, and bus bandwidth. These factors can introduce non-determinism into program execution. As per [11], [3], [21] the jitter induced by other hardware threads can be significant, and can make a program miss its deadlines. In the simple case, when two threads are executing alternately, one thread will displace lines in the caches belonging to the other line. This happens in our experiments.

Paolieri et. al.[11] focus on the important problem of bus bandwidth sharing, and proposes schemes for near

optimal sharing. They propose a *WCET* execution mode that delays bus accesses by a worst case duration but provides guarantees on the worst case time bound. In this case, the average memory access latency increases, but is predictable. They show that with a time penalty ranging from 2 to 20%, it is possible to make guarantees on the worst case execution.

Suh et. al. [21] provide an algorithm that uses a PID controller to stabilize the MIPS rate of a large set of embedded benchmarks. They use a PID controller that adjusts the voltage and frequency of a program to ensure a certain target MIPS rate. They observe that their controller is able to achieve the target rate, with close to optimal power consumption. This case requires the user to know the target MIPS rate before hand, and it also assumes that large parts of the program have the same MIPS rate. It is consequently oblivious of program phases, and thus has limited applicability.

### A.3 Power Management Jitter

There are two major power management schemes – dynamic voltage frequency scaling, and processor throttling. In the former approach, the frequency is decreased to save power. In the latter approach, the IPC is reduced by decreasing fetch and issue widths. They introduce jitter by slowing down computations in the code significantly.

One solution is to turn these schemes off while running time sensitive applications. However, this might adversely affect the health of the system. Hence, a better solution is to synchronize them with the program execution and apply DVFS or throttling at the correct points. We were not able to find any related work in this area.

## Appendix B
## Other Scenarios of Signal-Wait Jitter

### B.1 Case 2

Core 1 might wakeup another process after receiving the ipi, and subsequently wakeup the time sensitive thread. In this case, there will be multiple calls to the kernel(k_e and k_x events). This case can be detected by counting the number of k_e and k_x pairs. It will be greater than 1. Here also the jitter is the time difference between w_x and s_e and is accounted for as signal-wait jitter on core 1. If we decide to report the cause of jitter at a much finer granularity, then the jitter unit can record the number of (k_e,k_x) pairs. This will indicate the number of other processes that were run before our time sensitive task got swapped in. It can give the user a feel for the load on the system.

### B.2 Case 3

It is possible that the waiting thread might fire w_e on core 1, and then wake up on core 3 as shown in Figure 3. This can happen because, the kernel might be running another thread on core 1. If core 3 is free at that specific point of time, then the waiting thread might be sent to core 3. However, in this case, the jitter unit on core 3 knows that the process is waking up on another core since it will not find a corresponding entry in its wait buffer. Hence, it needs to broadcast the id of the thread, and get its details. Core 1, will send the entry (see Case 1) for the thread from its wait buffer. Please note that this entry will contain the time of the corresponding s_e event (see case 1). Using this information core 3 can compute the jitter. The opposite case, in which the signaling thread gets scheduled on another core is very rare. This is because of the overheads associated with process migration. However, if the waiting thread has been waiting for long enough, this overhead might not be significant.

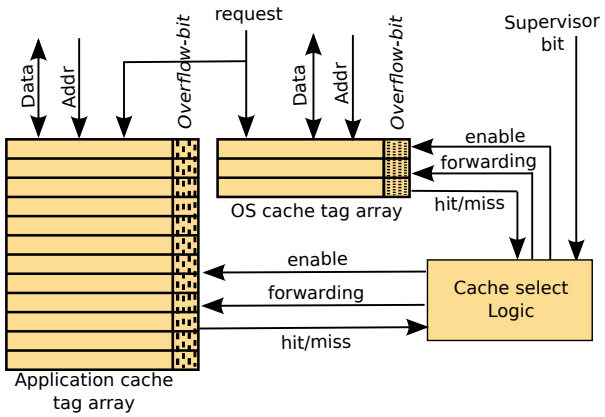



Fig. 14: Design of the OS cache

## Appendix C
## Implementation Details

### C.1 OS Cache

There are two pernicious effects of jitter – loss of computation time, and destructive interference in the memory system. The operating system (OS) cache tries to mitigate the latter effect. We propose to add a 64 KB cache at the L2 level dedicated for saving the cache lines that belong to the operating system. This cache can effectively reduce the interference between the application's lines and the OS lines.

We show a broad structure of the design in Figure 14. We do not add any extra structures at the L1 level because our simulations indicate that the gains are limited. However, at the L2 level, the gains are more promising. The first task is to identify the lines that belong the application and the operating system. We use the supervisor bit for this purpose. This bit is set to 1, when the operating system is executing. Any cache line (instruction/data) that is accessed at this time is directed to the OS cache. Likewise, when the supervisor bit is 0, we exclusively use the application cache.

However, this simple approach fails for shared pages between the application and the kernel. There are a

variety of situations in which the application and kernel need to communicate especially during read/write system calls, or when the kernel is initializing a part of user space memory. We propose to have all shared data in separate shared pages. For example, when a user level program is communicating with a block device, it can write its data to a shared page, which is also mapped into the kernel's address space. The kernel can then read the data from the shared page. All such shared pages, will have an additional shared bit set in both the page table and the TLB. Moreover, we propose to add an additional shared bit in every memory request. It is set to 1 if the cache line can possibly be shared between a user space process and the kernel, and it is set to 0, if there is no such possibility. For all shared pages, we arbitrarily choose to save them in the application cache. For all other lines, we choose the right cache based on the state of the supervisor bit at the time of the original memory access. Note that because of large latencies in an out of order pipeline, it is possible that by the time a request reaches the L2 cache, the supervisor bit has changed. Hence, we need to stamp every memory request with the state of the supervisor bit at the time it leaves the load-store queue.

We now optimize the combined cache for additional performance. We observe that if the footprint of a kernel invocation is large, then there can be a lot of capacity misses in the OS cache. Hence, we propose a scheme where the footprint of the OS can overflow into the application cache. Likewise, if a line is evicted from the application cache, it can overflow into the OS cache. We mark each line of a set with an overflow bit (marked in Figure 14). If the overflow bit is set to 1, and there is a miss, then the cache line might be presented in the other cache. The cache controller in this case sends a message to the other cache, and it checks for the line. If it is present, then it sends the line to the processor. Whenever a line is evicted from one cache, it is sent to the other cache. If the cache line is dirty, a copy of it is sent to the main memory too. This is done so that the other cache can evict it without writing it back to the main memory. The other cache will accommodate the overflowed line if the number of previously overflowed lines that have been accommodated in the set is less than half of its associativity. This condition restricts the amount of interference between the two caches, yet it wisely utilizes the available L2 space. A cache controller will evict the alien lines first to make space for its own incoming lines. This way, the alien line will always have the least priority.

The other subtle issue that we need to consider is the interaction of the combined L2 cache (application cache + OS cache) with the cache coherence protocol. It is important to note that, by design, a line is stored in at most one cache (L2 or OS). It is never stored in both the caches at the same time. Consequently, from the point of view of the directory protocol, the combined cache looks like one cache, and no additional coherence related actions need to be taken.

Gradually, all the overflow bits will start becoming 1, and we will always need to access the other cache. Hence, we propose to periodically flush the OS cache, and clear all the overflow bits in the application cache using a gang clear mechanism. We can do so at the beginning of an application context switch such that the process of writing back modified data to memory is most likely done by the time the OS begins to execute.

## C.2 Jitter Unit

### *C.2.1 Hardware Design*

Figure 15 shows the block diagram of the jitter unit. The jitter unit maintains two buffers (i) *wait buffer* and (ii) *lock buffer*. A *lock buffer controller* and *wait buffer controller* is responsible for managing the accesses to the lock buffer and the wait buffer respectively. Each element of these buffers stores the following fields: lock-id(64 bits), timestamp(32 bits), valid bit(1 bit).

Both the wait buffer and lock buffer are CAM (content addressable memory) arrays. However, to save power we have a serial lookup. They are addressed by the lock-id. There are three basic operations: *lookup*, *write*, and *modify timestamp*.

When a *lookup* for a particular lock-id is issued, the controller starts searching the corresponding buffer from the first location. On finding a matching entry in the buffer, the stored data is returned and the buffer location where the match occurred is cleared. We clear the buffer element on a successful match because there is never a case where the same lock-id is looked up more than once for the computation of the jitter. This observation simplifies the logic of the controller greatly. If this was not the case, we would have to store a list of free locations or manage the buffer as a circular queue. This would increase the complexity and the area of the jitter unit.

When a *write* is issued to the buffer, the controller again starts searching for free space from the first location and writes it in the first free location found in the buffer.

Lastly, the *modify timestamp* operation updates the current time in a buffer entry that contains the requested lock-id. This is handled in the same manner as a lookup but instead of returning the entry, the entry is updated with the new timestamp.

If the current operation completes successfully, the *success* line is set to high, otherwise it is set to low. A busy line is also used to indicate the status of the controller. When the controller is already servicing a request, the busy line is set to high. The busy line will be low when the controller is in the idle state.

The jitter unit is designed as a synchronous block in hardware in order to minimize area. Therefore, the jitter unit handles only one transaction at a time. A transaction is defined as a sequence of events which takes the jitter unit from the idle state through a set of

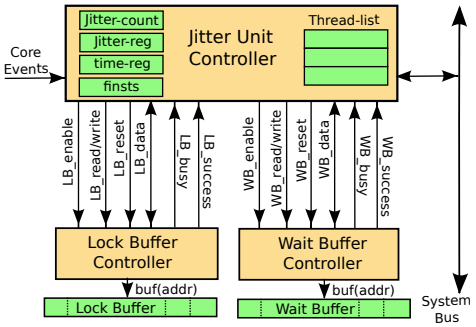



Fig. 15: Design of the jitter unit (One per core)

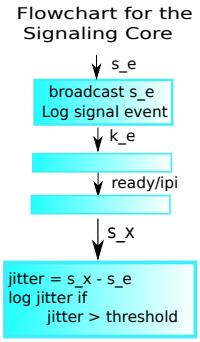



Fig. 16: Signaling core

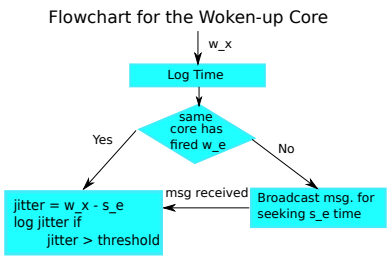



Fig. 17: Woken-up core

intermediate states and brings it back to the idle state. An asynchronous design of the jitter unit would lead to an increased area as in-flight transactions would have to be aborted and the jitter unit would be forced back to the idle state. Again, since the jitter unit would be sparingly occupied in a subframe, a synchronous design suits the requirement.

Since each jitter unit implements a distributed protocol, the controller does not distinguish between core and bus events. This is important since the jitter units exchange messages among themselves over the bus. Hence, the controller should respond to the events on the bus as well as events coming from the core like signal enter, wait exit and so on. The messages coming over the bus are latched when the jitter unit controller is processing a transaction.

### C.2.2 Jitter Unit Operation

The exact functioning of the jitter unit is now described for the signal-wait case. When a jitter unit sees a wait-enter event, the lock-id of the corresponding event is written into the wait buffer and the jitter unit on this particular core returns to the idle state. When a jitter unit on a remote core sees a signal-enter event, it broadcasts the event to all the other jitter units. On receiving this broadcast message, the jitter units on all the cores search their corresponding wait buffers. If a match for the corresponding lock-id (mentioned in the message) is found, the timestamp is stored against the lock-id in the wait buffer. If a match is not found, the jitter unit on the core ignores the message and returns to the idle state. The flowchart is shown in Figure 16.

As a result of this, the signal-enter(s_e) time for the given lock-id is stored with the jitter unit of the core on which the process went into the wait-state. Subsequently, the waiting thread will wake up and the jitter unit on the corresponding core will fire a wait-exit event (see Figure 17). There are 2 possible cases: (i) either the lock-id is found in the wait buffer or (ii) the lock-id is not found.

The first case implies that the process entered the wait-state on the same core. Therefore, the jitter unit has the timestamp of the associated signal-enter event. The second case implies that the process is waking up on a different core. The jitter unit now broadcasts the lock-id requesting the s_e time of this particular lock-id. Simultaneously, the jitter unit logs the s_e time in a temporary register – $time - reg$. All the jitter units lookup their wait buffer and the jitter unit that finds a match responds with the s_e time.

On obtaining the response from the remote core, the core on which the process will wake-up learns the necessary details to estimate jitter. The jitter unit on the remote core deletes the lock entry from its wait buffer. As shown in Figures 2 and 3, the jitter is the difference between the time of the wait-exit event and the s_e time stored against the corresponding lock-id. On encountering the signal-exit event, the jitter unit calculates the difference between the signal-exit and the previously seen signal-enter event. If the calculated jitter is above a certain threshold, it is added towards the overall jitter encountered in the current subframe by that core.

Additionally, the jitter unit maintains the thread-id of the current thread executing on the core in a special register, *jitter-reg*, time lost due to jitter since the beginning of a frame in a register, *jitter-count*, and the last used PCs of different time sensitive threads before a context switch, in a CAM array, *thread-list*. Along with these entries, we keep track of the number of instructions within a subframe in a separate register, *finsts*. All of these registers along with the buffers are collectively referred as the *jitter state* of the thread.

Since threads can potentially migrate across cores, we need to migrate a thread's jitter state across cores. Jitter state is created for a thread as soon as it is created. The core on which a thread is created writes an entry into its *thread-list* with its id. Before a kernel_entry event, the last used PC of a thread is written into its thread-list. Just after a kernel_exit, the jitter unit finds if the thread has migrated from another core. If this is the case, then the jitter unit broadcasts the thread id, and receives the jitter state on the bus. The jitter state is cleaned up after a thread exit. If the jitter state hasn't arrived by the time a w_x event arrives, then we need to solicit the value of s_e from another core (Figure 17) to compute jitter.

### C.2.3 Operations for Other Types of Jitter

Uptil now, we have covered signal-wait(sw) and signal(s) jitter. Let us consider the jitter associated with broadcast. In this case, there is one distinguished thread, which needs to broadcast a message to the rest of the threads. The case of pure broadcast jitter, is the same as pure signal jitter. We need to monitor the time between the b_e and b_x events.

For the broadcast-wait case, whenever a thread begins to wait, we need to create an entry in the wait buffer of the corresponding jitter unit. Akin to the signal-wait case, we need to timestamp the entry with the time of request of the b_e (broadcast-entry) event. Since there might potentially be multiple threads waiting, the b_e event has to be sent to all the jitter units, and all of them need to timestamp any relevant entries in their wait buffers. The rest of the protocol is the same as the signal-wait case.

For the case of lock-unlock jitter, we define two kinds of jitter events: (1) Lock-Unlock (lu) jitter, and pure Unlock (u) jitter. Unlike signal wait, it is possible for a lock to not be associated with a corresponding unlock. This is because the lock might not be contended, i.e., the lock might be free. In this case the lock entry event l_e need not be added to the lock buffer. The pthread library knows the status of the lock because every lock is associated with a memory address. If it contains a 1, then there is another thread that owns the lock, and if it is 0, then the lock is free. The futex based implementation of the thread library first uses atomic exchange primitives to obtain the lock. If it is not possible, then it issues a system call to put the current thread to sleep. At this point, we instrument the library to instruct the jitter to add a lock-enter(l_e) event in the lock buffer.

In this case, there has to be a corresponding unlock. Akin to the signal-wait case, we wait for the unlock enter (u_e) event that is sent to all the cores. If a core finds a lock entry in its lock buffer, then it time stamps it upon receiving the u_e event. The rest of the processing is the same as the signal wait case. The case for pure unlock jitter is the same as that of signal jitter.

Please note that for both signal-wait and lock-unlock jitter it is possible that there might be multiple pairs of processes that are executing signal-wait or lock-unlock operations simultaneously. If they are two different addresses, then our protocol does not have any problem. It always stamps every message with the lock address and matches it with the corresponding entries in either the lock buffer or wait buffer. However, if they are to the same address then also we can handle the situation, because every message contains the id of the thread that it is meant for. This is uncommon.

### C.2.4 Jitter due to Non-synchronization Events

We can use the same mechanism to account for jitter due to other sources such as the timer, I/O interrupts, or system calls. We need to track kernel entry and kernel exit events. Whenever a task enters the kernel due to a non-synchronization event, we log the event in the wait buffer. If the task wakes up on the same core, then we calculate the time elapsed after it was swapped out. If the triggering event was a system call, then we account for jitter only if the duration of inactivity of the time sensitive thread is more than some threshold, which can be defined by the user. Otherwise, we can account for the entire time as jitter. If the task wakes up on a separate core, then the jitter unit needs to broadcast the thread id, and handle the situation as in case 3 (see Section B.2).

## APPENDIX D
## COLLECTING JITTER TRACES

To the best of our knowledge, our paper is one of the first papers to exclusively look at OS jitter for multicore processors. Hence, we were not able to find any standardized methodology for simulating systems with jitter. The conventional approach for simulating systems with an underlying operating system is to use a full system simulator such as Qemu [32] or Simics [33]. These simulators can provide detailed traces at the instruction level for both application as well as OS activity. They use a simplistic set of models for basic hardware devices such as the hard disk, memory controller, network, and chipset. We do not consider this approach as the best method to simulate a system with OS jitter because it does not capture a vital aspect of real systems that we are interested in namely variability across runs. There is a sizeable variability across different runs of the same benchmark because of jitter inducing events. The main reason that full system simulators are not able to capture this is because they use a simplistic model of the entire system, and this model is not representative of complex server class machines.

However, we found some references from the traditional HPC community such as the proposal by De et. al. [34] for simulating OS jitter in MPI benchmarks. In this paper, we propose a model of simulating OS jitter inspired by [34]. Before outlining our approach, let us look at three aspects of simulating jitter:

1) We need to figure out the points within an application where it can be interrupted and other programs including the kernel can be run.
2) The second aspect is to ascertain the duration of a jitter event.
3) The last aspect is to collect a detailed trace (list of instructions and memory addresses) of the jitter inducing event, such that we can feed it to a conventional architectural simulator.

The first aspect, i.e., the points within an application that need to be interrupted can be found out by considering the most frequent types of jitter inducing events($> 97\%$) – synchronization events, and timer interrupts. We capture synchronization events and simulate a timer chip in our architectural simulator. Consequently,

we are able to correctly identify most of the points at which jitter needs to be introduced.

The next problem is to calculate the duration of the jitter. This is where we discourage using a full system simulator because most of the variability of application execution comes from the distribution of jitter duration in parallel multicore workloads. Alternatively, we collect jitter traces for each benchmark by instrumenting the 64 bit X86 GNU Libc new POSIX Thread (nptl) v2.10 library. For each synchronization event defined in Table 1, we add a function to record the time, event type, lock address, and thread id. We save the jitter records in a dedicated buffer in memory. These jitter traces are periodically dumped to a file. While dumping the results to the file, we turn off jitter monitoring. As previously mentioned, jitter can vary from system to system based on the actual settings. We collected our jitter traces on a 4 socket Intel Xeon server having four hyper-threaded cores (1.6 GHz, 2 MB L2 cache, 64 GB Main Memory) per socket running Linux kernel version 2.6.31. Before running a benchmark we ensure that our system is *properly tuned* (see Table 2 and [3]) such that the baseline OS jitter is minimized. In our experiments, we run each benchmark 10 times, and we observe a good degree of variability using the jitter traces collected using this procedure.

The next problem is to find out the nature of activity during the jitter inducing event. Since, we want to conduct a detailed architectural simulation, we need a detailed instruction level trace that has the following information per instruction executed – program counter, type of instruction, list of operands, and values of memory addresses (for load/store instructions). This instruction trace can be fed to an architectural simulation such that it can calculate the IPC and cache access statistics in great detail. It is not possible to collect this information in the previous step (collection of jitter durations), because a real machine does not provide this information. However, we shall use a couple of insights to make reasonable approximations.

We observe that any trace of instructions in a jitter inducing events starts with an interrupt, then the kernel loads the interrupt table and does some basic processing (top half). Subsequently, the kernel invokes the scheduler and tries to drain its work queue. If the time sensitive thread is at the head of the work queue, then it gets scheduled on the core. Here again, the kernel follows the same set of steps. It invokes a region of code that reinstates the TLB and the process control block. The main variability arises if other kernel tasks or user level tasks are at the head of the work queue. This is when the kernel opportunistically steals this opportunity to schedule other work. We observe that these tasks are relatively orthogonal to the nature of the overwhelming majority of interrupts (synchronization/timer). Consequently, we believe that it is sufficient to use the instruction trace of a jitter inducing event that has approximately the same duration as that of the jitter inducing event.

TABLE 8: Correlation of slowdown with jitter characteristics

| Splash (without *lu* and *radix*) | | | Parsec | | |
|---|---|---|---|---|---|
| col. 1 | col. 2 | col. 3 | col. 1 | col.2 | col.3 |
| -0.61 | 0.37 | -0.41 | 0.06 | 0.04 | -0.04 |

Consequently, we collect detailed instruction traces of kernel and daemon activity by running our setup on the Qemu [32](version 0.14) full system emulator. The processes on the kernel that occur most frequently are: init, kthreadd, kswapd, ksoftirqd/0... 16, kacpid, and kwatchdog. We collected 50 million entries. Along with the traces of kernel daemons, we also collect interrupt top/bottom half traces for I/O devices.

Lastly, we use an in-house architectural simulator to simulate the effects of OS jitter. The details of the simulator are given in Section 5.1. For any synchronization or timer interrupt we read the duration of jitter from one of our jitter duration traces, and find the nearest match (in terms of duration) from our kernel activity traces. The architectural simulator simulates the kernel trace and at the end restarts the time sensitive thread.

# APPENDIX E
# ADDITIONAL EXPERIMENTAL RESULTS

## E.1 Relationship with Jitter Characteristics

In Table 6 and Table 7, we show some more statistics about the jitter experienced in the Splash and Parsec suite of benchmarks respectively. In column 1, we show number of jitter events per millisecond. In column 2, we show the mean (arithmetic) duration of the jitter event. Column 3 is just column 1 multiplied by column 2. Finally, in the last column, we show the slowdown experienced by the application (also see Figure 10). Note that these numbers are for a specific run of the application, and they vary across different runs.

We observe that the number of jitter events per millisecond varies from 0.6 to 80 for Splash, and 0.14 to 34.47 for Parsec. We observe that roughly, once every 10 to 100 $\mu s$, there is a jitter event for some thread in the 16 thread application. Secondly, the average duration of the jitter event varies from 20 to 200 $\mu s$ for both the benchmarks. This means that the maximum load on the on-chip network is 1-5 messages (depending on the protocol state) per 10$\mu$s. This represents an extra overhead of 0.02% in terms of network traffic, which is negligible.

In Table 8, we try to correlate the values in column 1, 2, and 3 with the slowdown experienced. We compute Pearson's correlation coefficient [35]. A value of 0 means, that the values are uncorrelated. 1 means that the values are perfectly positively correlated (e.g., (1,2) and (3,6)), and -1 means that are perfectly negatively correlated (e.g., (1,2) and (2,1)).

We observe that *lu* and *radix* are outliers since they are kernel applications. They have very few synchronization

| app | (1) # jitter events per ms | (2) Avg jitter in $\mu$s | (3)=(1)*(2) | (4) slow-down % |
|---|---|---|---|---|
| *lu* | 0.69 | 239.92 | 165.03 | 0.25 |
| *barnes* | 79.96 | 19.64 | 1570.63 | 5.5 |
| *raytrace* | 35.69 | 130.21 | 4647.5 | 3.69 |
| *fmm* | 4.07 | 43.38 | 176.73 | 13.37 |
| *ocean* | 12.51 | 139.11 | 1740.26 | 35.61 |
| *water-nsq* | 21.83 | 67.35 | 1470.48 | 30.15 |
| *water-sp* | 12.99 | 93.88 | 1219.4 | 33.2 |
| *radix* | 0.6 | 256.76 | 153.2 | 0.62 |

TABLE 6: Jitter statistics for Splash2

| app | (1) # jitter events per ms | (2) Avg jitter in $\mu$s | (3)=(1)*(2) | (4) slow-down % |
|---|---|---|---|---|
| *facesim* | 1.56 | 198.93 | 310.3 | 10.01 |
| *ferret* | 0.45 | 29.3 | 13.24 | 1.65 |
| *bodytrack* | 1.2 | 48.77 | 58.56 | 1.62 |
| *x264* | 0.52 | 119.04 | 61.51 | 2.45 |
| *Raytrace* | 0.14 | 138.88 | 20.13 | 7.89 |
| *vips* | 0.65 | 15.43 | 10.03 | 0.9 |
| *dedup* | 2.97 | 282.34 | 837.74 | 7.03 |
| *fluidanimate* | 7.15 | 19.9 | 142.27 | 22.83 |
| *streamcluster* | 34.47 | 72.46 | 2497.71 | 4.53 |

TABLE 7: Jitter Statistics for Parsec

calls. Consequently, we eliminate them from this study. We observe that for Splash, the number of jitter events is negatively correlated with the slowdown.

We believe that this is because as the number of jitter events increase, the jitter gets distributed, and the slowdown is not so pronounced. There is a weak correlation between the average duration of jitter, and the total jitter experienced by all threads with the net slowdown in the case of Splash. However, we did not observe any such correlation relationships in Parsec. Since the slowdown is a very complicated function of the inherent nature of operating system jitter, structure of the critical path, and delays introduced by the underlying architecture, we found it very hard to make stronger predictions.

### E.2 Reactivity of the controller

We also plot the results for three values of reactivity: 1, 2 and 4 (see Section 4.1.6). The *reactivity* is the number of succeeding subframes across which the controller tries to eliminate the amount of accumulated jitter. Figure 18 shows the time overhead incurred in Splash2 and Parsec benchmarks with the effects of OS jitter and L2 evictions for various reactivities. We observe that jitter is compensated best for a reactivity of 1 and the time overhead gradually increases as the reactivity increases to 2, and then to 4.

Other than *water_spatial*, setting the reactivity to 1 is the best option. In the case of *water_spatial*, 2 is a marginally (by about 0.4%) better option. Secondly, for other than *water_nsquared*, 2 is a better option in terms of time than 4. A lower value of reactivity is a better option because it is possible that subsequent subframes might have some jitter of their own. When this is added to the jitter accumulated from previous subframes because of jitter spreading(see Section 4.1.6), the controller might not be able to totally nullify the jitter. There are two reasons for this. The first is that the controller might hit the limit in terms of the DVFS factor. Secondly, it might not have enough time to mitigate the jitter for the last few subframes.

### E.3 Jitter Unit Synthesis Results

We designed the jitter unit using the 90nm UMC standard cell library. The proposed jitter unit consumes 46166 $\mu m^2$ per core, of which, 65% (by area) is occupied by sequential logic (various buffers and status registers) and 31% by combinational logic (jitter-unit controller and buffer-controller). The remaining 4% is accounted for by inverter gates.

We scale the area of the jitter-unit from $90nm$ to $32nm$ technology using the scaling techniques mentioned in [23]. 32 $nm$ is 3 generations ahead of 90 $nm$ technology and with each generation, we make a conservative assumption that gate area reduces by 43%. Thus, in 32 $nm$ technology, the jitter-unit will require 8550 $\mu m^2$. Similarly, from one generation to another generation, we assume that the gate delay decreases by 10%. Thus, the maximum delay of the jitter-unit will go down from 850 $ps$ in 90 $nm$ to 620 $ps$ in 32 $nm$ technology. A die implemented in $32nm$ technology usually contains upto 16 cores. Thus, the overall area required by all the jitter-units is 136800 $\mu m^2$. As previously mentioned, the jitter-units use the existing on-chip interconnect to communicate between themselves and hence no additional area is required for the inter-connect between the jitter-units.

## APPENDIX F
## OTHER SCHEMES TO MITIGATE JITTER

### F.1 Prefetching

We propose a novel scheme based on prefetching to reduce the destructive interference in the memory system experienced after a context switch. We tried to modify standard prefeteching algorithms to come up with a variant that works the best for a scenario with OS interference. Our proposed algorithm is as follows.

We propose two additional structures – a prefetch buffer and a jitter duration predictor. The prefetch buffer maintains a list of cache lines belonging to either the application or the OS. The duration predictor predicts the time of the next context switch using a method based on moving averages. It maintains the average duration of an OS execution epoch, and an application execution epoch. An *epoch* is a continuous run of the application or the OS without an intervening context switch. It uses the mean value as the best estimate of the current epoch (OS/application). $\tau$ cycles before the end of the epoch, it

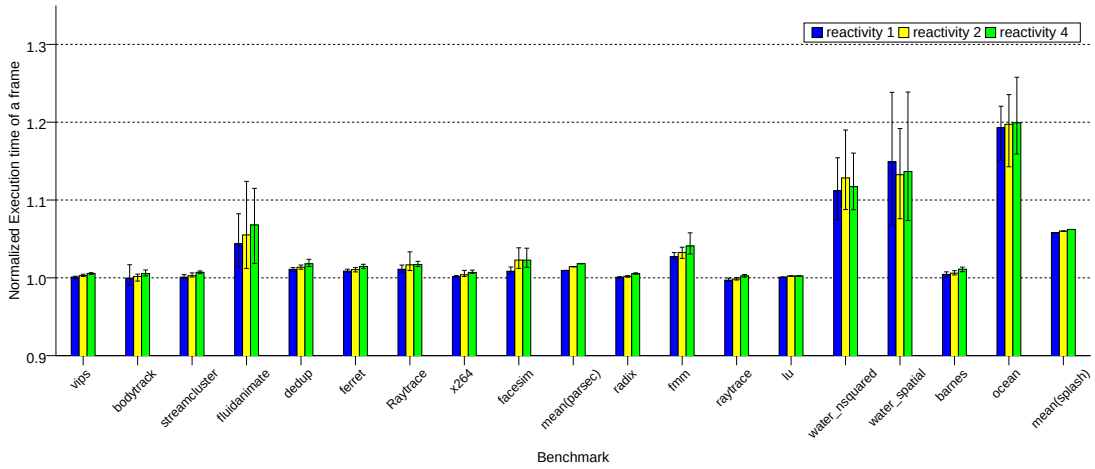



Fig. 18: Time overhead for different reactivities (Splash2 and Parsec)

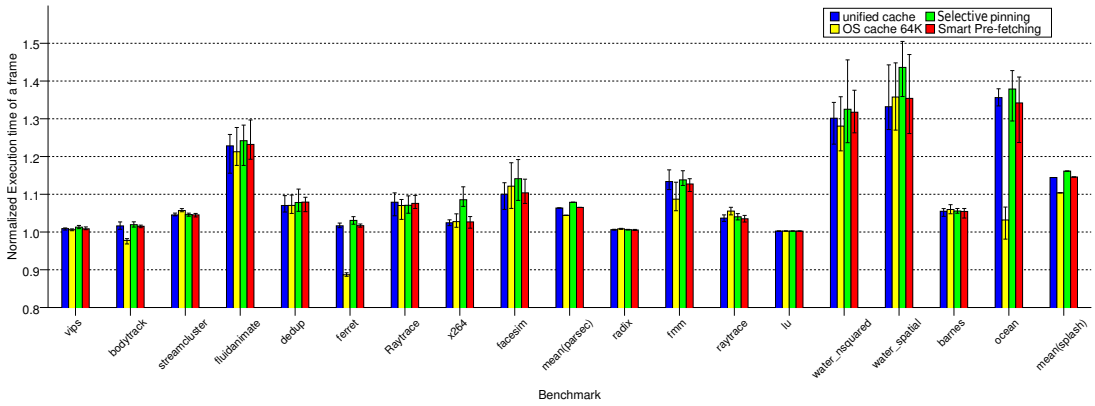



Fig. 19: Time overhead for other proposals (Splash2 and Parsec)

prefetches the lines present in the prefetch buffer to the L2 cache.

Let us now explain the operation of the prefetching logic. First, we define the notion of an *active* cache line. We denote a cache line as *active* if it has been accessed in the last $\mathcal{N}$ cycles in the L2 cache. We arrived at the number, $\mathcal{N}$, through extensive simulations. To maintain this information, we need to add a timestamp to every cache line in the tag array. When the operating system evicts an active application line from the cache, the prefetching logic inserts the line into the prefetch buffer. If the line is modified, then it writes it back to memory. $\tau$ seconds before a context switch, it brings the lines in the prefetch buffer back to the main cache. We use the same approach for the scenario in which the application code evicts OS lines. The prefetch buffer is a linear SRAM array.

While bringing back a line from the prefetch buffer to the L2 cache, we must make sure that the alien line is not evicting an active line belonging to the currently executing code. If we do not find any appropriate candidate lines that can be evicted, then we do not prefetch the line. We choose $\tau$ to be equal to 10% of the expected epoch duration, and $\mathcal{N}$ to be 256 times the average duration between L2 cache accesses.

### F.2 Selective Pinning

Most modern processors use a version of LRU (Least Recently Used) as a cache eviction policy. This scheme works well since most applications display temporal locality. However, this scheme does not work well between the application and operating system. When the execution of the operating system starts, the timestamps of all the cache lines belonging to the application are "recent", whereas the OS lines are marked as "old". This is unfair since the OS needs to favor its own cache lines. We propose to rectify this using a variation of pinning. Note that we arrived at this heuristic after trying out many standard methods such as pinning all the OS cache lines, most recent OS lines, and most recent application lines.

We devised a novel scheme called *Selective Pinning*, which decreases the priority of a cache line based on the amount of time left in the current epoch. During the initial 95% of the execution, the priority of a cache line belonging to the current privilege level is higher compared to other lines. Towards the end of the epoch, the priority of other cache lines is more than the lines belonging to current privilege level. For example, towards the end of an OS epoch, application lines begin to have a greater priority, and vice versa. To implement

this scheme, we tag each cache line with a bit indicating the privilege level (1 for application, and 0 for the OS). Instead of a discrete version as proposed, we tried with other continuous versions of this scheme that gradually modify the priority. However, the simulation results favored this scheme.

### F.3 Evaluation

Figure 19 shows the experimental results of the proposals mentioned in Section F.1 and F.2.

We observe that *selective pinning* performs worse than the *unified cache* configuration in all of the 17 benchmarks. The results are very adverse in the case of *water_spatial* where the difference is over 10%. The main reason for this is that pinning is not able to correctly capture the temporal locality of accesses. If we pin a line, we are betting on the fact that the line will be accessed in the near future, and the line that was evicted will not be required in the near future. However, our scheme is not able to adjudge these probabilities very efficiently.

*Pre-fetching* does not offer significant advantages over the *unified cache* either. However, we do not observe any significant slowdowns due to this method. Refining these methods to yield appreciable speedups is a part of future work.