Appears in ***13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII)***

Adapting to Intermittent Faults in Multicore Systems

Philip M. Wells Koushik Chakraborty Gurindar S. Sohi

Computer Sciences Department
University of Wisconsin, Madison
{pwells, kchak, sohi}@cs.wisc.edu

Abstract

Future multicore processors will be more susceptible to a variety of hardware failures. In particular, *intermittent faults*, caused in part by manufacturing, thermal, and voltage variations, can cause bursts of frequent faults that last from several cycles to several seconds or more. Due to practical limitations of circuit techniques, cost-effective reliability will likely require the ability to temporarily suspend execution on a core during periods of intermittent faults.

We investigate three of the most obvious techniques for adapting to the dynamically changing resource availability caused by intermittent faults, and demonstrate their different system-level implications. We show that system software reconfiguration has very high overhead, that temporarily pausing execution on a faulty core can lead to cascading livelock, and that using spare cores has high fault-free cost. To remedy these and other drawbacks of the three baseline techniques, we propose using a thin hardware/firmware layer to manage an *overcommitted system* — one where the OS is configured to use more virtual processors than the number of currently available physical cores. We show that this proposed technique can gracefully degrade performance during intermittent faults of various duration with low overhead, without involving system software, and without requiring spare cores.

Categories and Subject Descriptors B.8.2 [*Performance and Reliability*]: Performance Analysis and Design Aids

General Terms Design, Performance, Reliability

Keywords Intermittent Faults, Overcommitted System

1. Introduction

The components of future multicore processors will become less reliable as technology scales, because individual devices are increasingly susceptible to a variety of hardware faults [6, 8, 10, 12, 34, 36]. In particular, technology experts warn about an increase in *intermittent* faults — faults which occur frequently and irregularly for a period of time, commonly due to process variation or in-progress wear-out, combined with voltage and temperature fluctuations [6, 8, 12, 13]. These are in addition to *transient* faults (or soft errors), typically caused by particle strikes, and *permanent* faults, which occur repeatedly after a device sustains irreversible damage.

Hardware reliability techniques have shown great promise at tolerating faults, i.e., at hiding the effects of faults from the system- and user-level software running on the hardware [4, 16, 19, 20, 27, 30, 31, 37, 40]. Such techniques, however, have weaknesses along one or more axis of *fault coverage*; *overheads* from time, power, and area; and *circuit complexity*. As devices become more unreliable, the ways in which faults manifest are likely to increase, with a consequential increase in the complexity and overhead of the techniques to tolerate the faults.

Intermittent faults present further challenges for designers. Unlike transient faults due to single-event upset (SEU), intermittent faults occur in bursts that can last from several cycles to several seconds or more. While it may be possible for complex hardware and/or software schemes to deal with a variety of intermittent faults on their own, we believe that such schemes will be neither practical nor desirable. We must, therefore, explore additional techniques that can enable circuit-level schemes to be more effective against these faults. In particular, we believe that the ability to temporarily suspend program execution on a core that is sustaining intermittent faults will be an effective ploy for 1) reducing several of the factors contributing to the faults in the first place, 2) simplifying system design by reducing the fault coverage requirement, and 3) aiding in the diagnosis of permanent circuit damage.

Naively suspending a processing core is not a common practice because it is not transparent to software and can lead to serious system-level consequences. Fortunately, multicore processors provide unique opportunities for enabling techniques to *adapt* to the dynamically changing resource availability created by intermittent faults. We present a qualitative and quantitative comparison of three of the most logical adaptation techniques, 1) pausing execution on the faulty core without notifying the OS, 2) using spare cores, and 3) asking the OS to stop using the faulty core, and demonstrate their different system-level effects. To remedy several drawbacks of these three, we propose a fourth technique: using a thin hardware/firmware layer to manage an *overcommitted system* — one where the OS is configured to use more virtual processors than the number of physical cores [46].

Before exploring our results in detail (Section 5), we discuss the causes of intermittent faults and our major assumptions (Section 2), explore the qualitative properties of the four adaptation techniques (Section 3), and provide details of our methodology (Section 4).

2. Background: Intermittent Faults

Intermittent hardware faults are hardware errors which occur in bursts for a period of time, commonly due to process variation or in-progress wear-out, combined with voltage and temperature fluctuations (often called *PVT* variations), among other factors [6, 8, 12, 13]. These variations can result in timing errors even when operating conditions are well within the specified noise margins.

Because intermittent faults are affected by a large number of factors, the duration of bursty faults can vary across a wide range of timescales. For example, voltage fluctuations are typically short-

ASPLOS'08, March 1–5, 2008, Seattle, Washington, USA.

lived, on the order of several to hundreds of nanoseconds [8,22,33]. Temperature fluctuations alter a device's timing characteristics over millisecond to second time scales [32]. Different software phases, which can change on the order of 100ms to several seconds [35], can exercise different components of a core, activating different intermittent faults. Finally, as wear-out progresses over the course of days [40], it can cause intermittent faults to become frequent enough to be classified as permanent [12].

Many uncertainties remain regarding the occurrence of intermittent faults in future technologies. In this paper, we make three primary assumptions regarding these faults. Below we examine the insights that lead us to believe these assumptions are reasonable.

1) Bursty intermittent faults will occur frequently. While the exact rates of various faults are not certain for future processors, current technology trends clearly indicate that even the design of commodity processors will be greatly affected by these faults. First, wear-out failures are expected to become much more frequent [7, 8, 34], but devices typically do not fail suddenly, they display intermittent behavior for a period of time beforehand [12, 13, 40]. Second, continued device scaling will result in increased PVT variations, increased cross-talk, and decreased noise margins [7, 8, 10, 12, 13, 34], all of which lead to increased susceptibility to intermittent timing faults.

The fault rates we choose to examine in this paper are in the range of 0.1% *Fault Duty-Cycle* and up, meaning each core is affected by a fault 0.1% of the time or more. We study such high rates for several reasons. First, by studying and proposing solutions that remain effective at such rates, we help process researchers understand what frequencies of hardware faults can be tolerated by higher layers in the system. Second, it is unclear whether intermittent faults between multiple cores will be correlated, though our evaluations assume course-granularity failures are independent.[1] High rates make it likely that multiple cores will fail at the same time, approximating a period of correlated faults. Finally, these rates are well beyond the expectation for current systems, but not beyond the public considerations of industry technologists [6].

2) Practical circuits cannot mask all intermittent faults. While many techniques for tolerating various faults have been proposed [4, 16, 19, 20, 27, 30, 31, 37, 40], the ways in which faults manifest are likely to increase as devices become more unreliable. This will lead to a continued increase in the complexity and overhead of the techniques to tolerate the faults. We believe circuit, and higher-level, techniques will thus be employed to reduce the frequency of intermittent faults, but *cost-effective* techniques are unlikely to completely eliminate these faults, or prevent their occurrence from being noticed by system or application software.

For example, techniques such as Razor [16] can detect and correct many timing related faults until the timing errors become too large. After that point, techniques like Razor will be forced to either fall back on another, much more complex and higher overhead reliability technique, or adopt a simpler policy of suspending the use of a core while conditions stabilize.

3a) Suspending use of a core . . . reduces factors causing faults. Suspending the use of a core cannot repair manufacturing variations or in-progress wear-out. However, temporarily suspending computation on a core *will* cause temperature and voltages to stabilize, reducing the further occurrence of any intermittent faults caused by these two major factors.

3b) . . . reduces faults. Suspending the use of a core when a burst of faults begins, or is expected to occur, can improve the overall reliability of the system. Due to the factors influencing intermittent faults, correlated faults *within* a core or other localized area are very likely. Thus, the kinds of events that are most challenging for existing techniques to protect against, e.g., multiple concurrent faults or faults affecting critical structures, are most likely to occur together during temperature, voltage, or other fluctuations. All reliability techniques have a certain probability of manifesting unprotected errors to higher levels in the system. By reducing the number of faults they must protect, especially the ones they are least likely to protect, the number of faults they miss is reduced.

Current high-availability systems already take a similar approach by having service technicians replacing chips when correctable intermittent faults begin to occur [12]. However, the granularity of failure in a multicore (*portions* as opposed to an *entire* chip), and the increasing frequency of these faults even for commodity processors, make chip-level replacement undesirable.

3c) . . . is likely to be useful for other purposes. Several proposals call for fine-grained reconfiguration of a core's components (e.g., [9, 37]), or match a program's requirements to a particular core's degraded capabilities, (e.g., [21]). We believe that the ability to suspend execution on a core, without significantly impacting the rest of the system, makes these techniques more feasible. We expect additional uses for temporarily suspending computation will be discovered as architects consider the viability of such a policy.

3. Adapting to Intermittent Faults

Naively suspending program execution on a core is not a common practice because it is not transparent to software and can have serious system-level implications. Fortunately, multicore processors provide unique opportunities, including inherent redundancy, low on-chip latency, and high bandwidth, which enable several techniques for adapting to the temporary loss of one or more cores. In this section, we discuss three such techniques which represent the current state-of-the-art, and propose a fourth technique to remedy serious drawbacks in each of the first three. In Section 5, we present a quantitative comparison of all four techniques, considering throughput, effects on latency-critical applications, fairness, and overheads at different fault rates.

Throughout this discussion, we refer to the chip's physical cores simply as *cores*. We refer to the software-visible processing units as *virtual processors* or *VCPUs*. In many cases, the two are equivalent. However, the hardware/firmware may choose to expose more or fewer virtual processors to software than there are physical cores, or it may transparently reassign a VCPU from one core to another. For simplicity, we typically refer to the lowest software layer as the *operating system* (OS), which could be replaced with *hypervisor* throughout with no loss of generality.

3.1 Existing Adaptation Techniques

Technique 1: Pause Execution The first technique we examine for suspending the use of a core is to just pause the execution of instructions for a period of time. As shown in Figure 1(a), when a core (C2 in this case) sustains an intermittent fault, the microarchitecture pauses the execution of instructions from the virtual processor assigned to that core (V2).

Pausing execution is the simplest technique we examine, and has been used, in a uniprocessor at least, for thermal management [18]. In a multicore, other cores continue to execute instructions, thus pausing execution on one core will not drastically affect the other cores as long as they do not attempt to communicate with the thread assigned to the paused core. If communication is present, however, pausing one core can cause a cascading effect, livelocking other cores.

This technique is not *fair*, because threads scheduled on the paused virtual processor are starved, and it can similarly impact the latency of critical applications. We would expect to observe low

[1] This assumption is only relevant to Section 5.3.

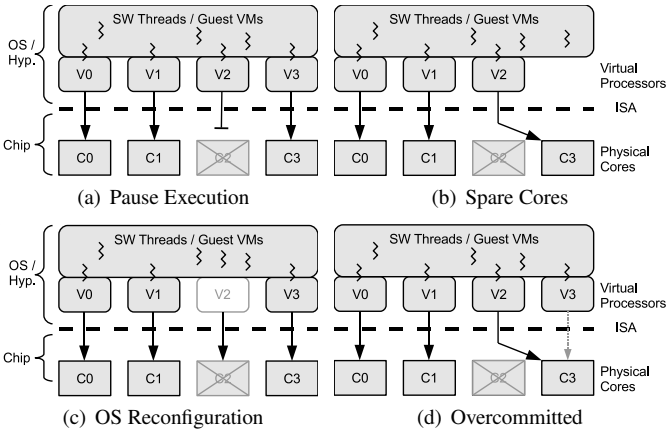

Figure 1. Core Suspension Techniques

throughput for workloads where threads frequently communicate, but for faults of short duration, this technique may be adequate for some applications.

Technique 2: Spare Cores Unlike pausing, setting aside one or more cores as spares is expected to have little impact on software during a fault. For example, an eight-core chip might only expose seven cores to the OS. During a fault, the chip, using a hardware/firmware layer, can transparently remap the affected virtual processor from the faulty core to the spare. (We discuss recovering the state of the VCPU in Section 3.3.) Core sparing is depicted in Figure 1(b).

Hot (powered up) spares are appropriate for short duration intermittent faults as circuit techniques can reduce leakage power when the core is not needed [43]. Since the performance degradation is negligible during a fault (as long as the number of faults do not exceed the number of spares), spare cores are also effective for long-duration or permanent faults. Partly for these reason, spares are used in real systems (though to our knowledge, for permanent faults only) [3, 39].

The major drawback of setting aside spare cores, especially for commodity processors, is the high overhead of *not* using these cores during fault-free execution. In addition, this technique cannot tolerate more concurrent failures than the number of spares without an additional fall-back mechanism.

Technique 3: OS Reconfiguration A third possible technique is to ask the OS (or hypervisor) to reconfigure itself to only use the remaining fault-free cores. This technique is depicted in Figure 1(c), where the de-configured virtual processor is not assigned any software threads or guest virtual machines to run. Some current OSs (such as Solaris) and hypervisors (such as those that run on the IBM zSeries) already contain this functionality [2,42]. For other systems, this technique requires intrusive software modifications.

Software reconfiguration can take several milliseconds, and can cause high overheads for faults of short duration. But the performance of the system should gracefully degrade once reconfiguration is complete, since the OS retains responsibility for scheduling threads, maintaining fairness, and achieving low latency for critical applications.

On the surface, this technique also appears to eliminate the need for hardware adaptation mechanisms. Unfortunately, that is not the case, since current operating systems requires the faulty core, and all other cores, to operate correctly until reconfiguration is complete [28,42]. For the evaluations in Section 5, we utilize our proposed technique (discussed next) during reconfiguration, though it is not needed once reconfiguration has taken place.

3.2 Utilizing an Overcommitted System

A qualitative look at three existing techniques for suspending the use of a core has revealed several deficiencies: fairness and latency concerns, along with the possibility of cascading livelock; high fault-free overhead and the need for a fall-back mechanism; and OS-intrusive modifications plus the need for advanced notice of an upcoming fault.

Technique Overview To alleviate these drawbacks, we propose a fourth technique which uses a thin hardware/firmware layer to abstract the details of fault management from the system software, while presenting a view of continuous, fully-functional, reliable operation. Such abstraction is achieved during periods of intermittent faults by building on our prior work on an *overcommitted system* — one where the system software is configured to use more processors than the number of currently available cores [46]. This technique operates beneath the ISA, making it applicable to all system software that can be loaded on the machine.

The function of the hardware/firmware layer is to virtualize the cores. That is, the system software is not allowed to directly control the *physical cores* implemented on the chip, but rather control the *virtual processors (VCPUs)* that are exposed to it. The hardware/firmware layer then manages the mapping of VCPUs to cores, such that a given VCPU, unbeknownst to the system software, can be quickly migrated to a different physical core, or briefly paused. The operation of this layer is similar to, but much simpler than a traditional Virtual Machine Monitor (VMM).

In an overcommitted system, two (or more) OS-visible VCPUs must share a single physical core during a fault. Figure 1(d) shows an example of using this technique, with virtual processors V2 and V3 sharing core C3. V2 is currently executing, while V3 is not, but they can frequently switch to avoid the issues associated with the generic *pause* technique. The VCPUs that are co-assigned are rotated to achieve fairness; for example, at some point, V2 may have C3 to itself while V3 and V0 share C0.

We use hardware spin detection to facilitate overcommitting unmodified Solaris [26, 46], since a VCPU that is not currently running could be holding a kernel lock or be the recipient of a cross-call. Spin detection preempts requesters spinning on the lock, or initiators waiting for acknowledgment of the cross-call, in favor of VCPUs that are performing useful work. Spin detection is *not* required for correctness, as long as the hardware/firmware periodically forces a VCPU switch. Spin detection is, however, an important performance optimization.

Hardware/Firmware Complexity This technique involves modest hardware/firmware complexity. Required features involve a mechanism to context switch a virtual processor, a VCPU to core mapping table, spin detection hardware, and control logic.

In our model, transferring a VCPU from one core to another involves first moving all of the processor state, including visible state and control registers, into the caches. The state is then restored on another core (or later on the same core), allowing the coherence protocol to transparently migrate the data when necessary. To reduce complexity, we assume that this functionality is implemented in firmware/microcode using loads and stores to a reserved portion of the physical address space. This support for transferring state is similar to that contained in current products [1, 44].

A table mapping VCPUs to their currently assigned cores is necessary both for scheduling decisions and for interrupt delivery. We assume this small, infrequently accessed table is implemented in hardware, and is hardened or replicated to protect against faults.

Spin detection hardware is helpful with unmodified Solaris. However, virtualization-aware OSs, or ISAs that suggest the use of a particular instruction to indicate spinning or idle processors

(similar to X86's `hlt` instruction) can eliminate this added complexity [2, 45].

Finally, we need control logic with inputs from the fault detection mechanism, spin detection hardware, and mapping table. This logic needs to perform simple scheduling decisions, direct the migration of virtual processors, and maintain the mapping table. We assume this simple logic is implemented in hardware.

Overall, this is a modest amount of complexity, though certainly more than is required for the pausing technique. However, all of these components except spin detection are already required in order to use spare cores.

ISA Transparency By placing control over the use of faulty and non-faulty cores below the ISA, the abstraction of fault-free operation occurs transparently to both the OS and a traditional hypervisor such as VMWare or IBM's Power5 Hypervisor. Such a model allows chip manufacturers to ship a chip that is expected to experience intermittent faults, but will continue to operate correctly regardless of the system software installed on the machine. The model provides several benefits for chip makers: first, the burden of correct hardware operation remains with the hardware vendor, not the system software vendor; second, the new chip automatically works with products from multiple system software vendors, and with legacy system software as well; and finally, as we will see in Section 5, placing control of faulty cores beneath the ISA allows some of the functionality to be implemented in hardware, making it easier to quickly adapt to frequent changes in hardware configuration.

3.3 Other Issues

We make two additional assumptions about hardware detection and recovery mechanisms to frame our continued discussions. First, three of the mechanisms require that the virtual processor executing on the suspended core be moved to a different core. Though recovering the state from a suspended core may be possible in certain circumstances (e.g., [28]), it is clearly infeasible for others. Instead, we assume that the fault recovery mechanism periodically creates checkpoints, similar to [37], [25], or [41]. The checkpoints are stored into the cache every 10k cycles, and on I/O, and are consistent across the on-chip cores [25, 41].

Second, we assume that circuit-level techniques exist within a single core for detecting and recovering from many simple faults, whereas upon detecting a rash of intermittent faults on a core, the circuit mechanisms initiate a rollback to the previous validated checkpoint and then begin the adaptation mechanisms. The use of Dual-Modular Redundancy (DMR), or triple redundancy (TMR), as a detection and recovery mechanism is also be possible. Since we are primarily interested in the effects on software, the results of this paper would remain unchanged if one considers DMR cores to form one logical processing core, and then performs the adaptation techniques only on the logical core.

If using TMR, the temporary loss of only one of the three redundant cores might still allow the continued use of DMR on the remaining cores. Though several nice properties of TMR disappear, including extremely high coverage and forward error correction, the continued coverage may be sufficient, reducing the amount of time it may be necessary to invoke core suspension mechanisms.

4. Experimental Methodology

4.1 Simulation

For the experiments in Section 5, we use Virtutech Simics [29], an execution driven, full-system simulator which functionally models a *SunFire 6800* server in sufficient detail to boot unmodified operating systems. We use Simics as a functional simulator only, and model timing using Simics MAI with our own cycle-accurate processor and memory hierarchy module.

We model both an OOO core and, to reduce simulation time, an in-order core for all experiments of 100ms or longer. We model each OOO core as a 2-wide, 128-entry window core at 3 GHz. The in-order core is a simple blocking model. The chip exposes eight cores to the OS in most experiments. Each core contains split 16k, 2-way I&D caches, and a unified 512k, 4-way private L2. We also model a 16MB, 16-way, shared L3 that is exclusive with the L2s. Cores maintain coherence via a MOSI directory protocol over a point-to-point interconnect with an average 10 cycle latency. The L2 directory uses shadow tags, which are co-located with each L3 bank. Main memory is 350 cycles load-to-use, with 40 GB/sec of off-chip bandwidth. These microarchitecture parameters of the cores and caches have little practical impact on our results.

For experiments which use processor virtualization, we evaluate a thin virtual-machine layer assuming low-level firmware with hardware support. We do model the overhead of firmware execution to migrate VCPUs. This task is performed by storing the running VCPU's state in a portion of cacheable physical memory and loading it later from the same or a different core. The state can be transparently migrated to other cores using the on-chip coherence protocol. Swapping VCPUs on a core requires several hundred cycles to store and then load the large SPARC V9 architected register state, and migrating costs up to 1000 cycles. We use the *Spin Detection Buffer* from [46].

In all simulations, we pause all cores for 10k cycles (3.3μsec) upon initiation of fault recovery to roll back to the latest verified checkpoint and account for the work lost.

OS Reconfiguration To perform OS adaptation, we instruct Solaris to *unconfigure* one of its eight virtual processors, CPU4. In our simulations, we do this by sending an interrupt to a second processor, CPU3. As the interrupt is executing on CPU3, we force it to call `sbd_ioctl()` with the necessary arguments to unconfigure CPU4. The function and arguments are the same as would be called by the command `cfgadm -C unconfigure CPU4`, but the interrupt mechanism allows us to call this function at arbitrary points without the overhead of the command. Note that the Solaris `psradm` command, which can take CPU4 'off-line' is insufficient, as the processor is still required to process cross-calls. Also note that due to limitations in the Simics functional model, we are unable to perform this experiment with newer versions of Solaris, or to perform the analogous experiment of reconfiguring a CPU. We use the overcommitted technique as the fall-back mechanism until the virtual processor is unconfigured.

Spare cores Due to our methodology, comparing runs using separate commercial workload checkpoints with different numbers of OS-visible VCPUs is impractical. Thus, for the throughput experiments, we model spare cores using the overcommitted technique with oracle spin detection and without charging overhead for storing, migrating, or switching VCPU state. We *do* properly simulate a seven processor system with our microbenchmark for latency and fairness experiments, since our microbenchmark (discussed below) is very regular and is dominated by user code.

4.2 Workloads

We use several workloads for these experiments. *vortexMIX* is a simple multiprogramming workload consisting of 8 copies or *255.vortex* from SpecINT2000 running reference inputs. *OLTP* is a TPC-C-like workload using IBM's DB2 database. The database is scaled down from TPC-C specification to about 800MB and runs 192 concurrent user threads with no think time. *Apache* and *Zeus* are static web servers driven by the Surge client. We do not use any think time in the Surge client. *pmake* is a parallel compile of

Measuring Throughput We evaluate workload throughput in Section 5.1. For these experiments, one core experiences a detected fault at the beginning of execution; simulations are then run for a range of times from 100μs to 1 second.
Measuring Latency and Fairness For both the latency and fairness experiments in Section 5.2, we use our microbenchmark and simulate a single 10ms fault beginning at 100μs of simulation, and then run for 11ms. The spare core experiments have seven threads and seven VCPUs, with eight threads and eight VCPUs for the rest.
Measuring Overall Performance Impact Determining the overall impact of intermittent faults requires accounting for periods of fault-free execution as well. We run several experiments with faults randomly occurring at a particular rate. The fault duration is fixed in each experiment, but the inter-arrival time of faults is sampled, independently for each core, from a normal distribution of moderate variance ($\frac{\mu}{\sigma} = 10$). We randomize in-progress faults and inter-arrival latencies at the beginning of simulation, and run enough randomized trials to achieve a proper distribution of long faults with only 1sec simulations. However, we cannot properly setup the system software at the beginning of simulation to *already* be affected by an in-progress fault, and thus the results reported for the pause scheme are optimistic for the longer fault durations. Using much longer trials would require computationally intractable simulation for these experiments.

Table 1. Experiment Descriptions

PostgreSQL using GNU make and the Sun Forte Developer 7 C compiler. We do not include serial phases. *artOMP* is 330.art_m from the SpecOMP2001 suite, using reference inputs, and warmed up and running in steady-state. Due to workload variability, we simulate multiple runs and report average results. For readability, we omit confidence intervals from the graphs, but we run sufficient trials and observe small enough variance to keep the 95% C.I. within 10%, and typically much less. Furthermore, small performance variations have no effect on the qualitative results of this paper. We use a microbenchmark in Section 5.2, which consists of one thread per processor, where threads each execute short CPU-bound transactions and have no communication. We use *committed user instructions* as our metric for work in all experiments. User commits has been shown to correlate well with other 'work' metrics, such as workload transactions [47].

5. Quantitative Analysis

Intermittent faults are caused by a variety of factors, and typically last for a range of durations (see Section 2). In this section we present a quantitative analysis to understand the implications of different fault rates and durations on the four adaption techniques.

We focus on two kinds of experiments. First, we inspect the system behavior in detail during a fault by measuring throughput, latency and fairness. After we understand these implications, we inspect the overall impact of these faults, including fault-free execution, for a wide range of fault durations and frequencies. The three types of experiments we perform are described in Table 1.

5.1 Throughput During a Fault

In this section, we demonstrate the throughput of all four techniques *during* intermittent faults of various durations, discussing each in turn. The line at $\frac{7}{8}$ in throughput graphs represents the expected best-case slowdown of losing one core.

Pause Execution Figure 2 shows the throughput of each benchmark when pausing execution for faults of various durations. For

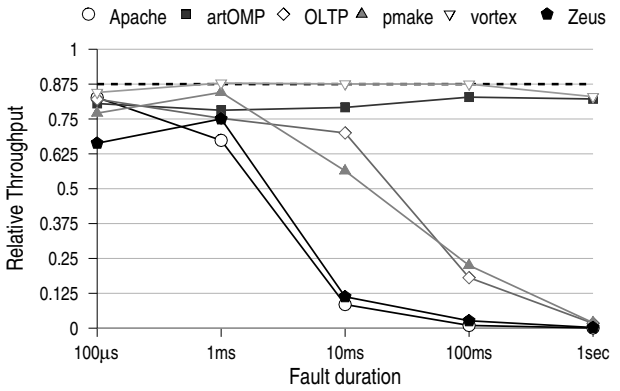

Figure 2. Throughput of Pausing Execution During Faults of Various Duration

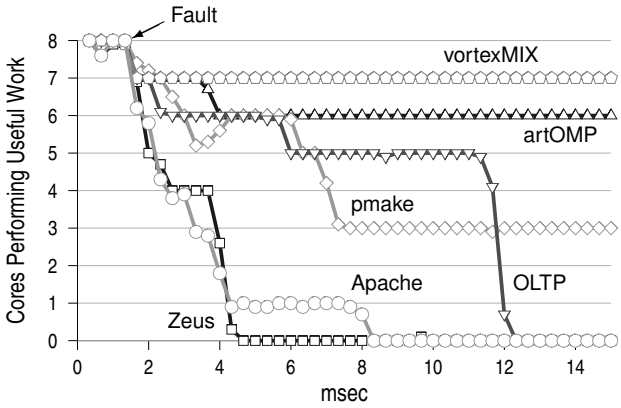

Figure 3. Cascading Livelock of Pause Scheme

the shorter 100μs and 1ms faults, all workloads observe throughput within 25% of the best case. Across a range of fault durations, *vortex* continues to have throughput similar to the best case, while *art* is only slightly lower than that. The commercial workloads, on the other hand, which have significant OS activity and communication between cores, observe much lower throughput for faults of duration greater than 1ms — even approaching *zero* throughput for 100ms and 1sec faults.

Figure 3 helps explain this throughput loss for longer faults. This figure shows the first portion of a trace of the number of cores performing useful work during every 0.3ms of a 100ms fault. We define *useful work* for each core as whether or not any user instructions were executed in each 0.3ms period, and sum this boolean value over all eight OS-visible cores. For all workloads, the number of cores performing useful work immediately drops to seven (or lower) after the fault. Vortex, with eight independent processes, remains at seven for the duration of the fault. For *artOMP*, a second core stops performing work after 2ms because it has blocked waiting on a TLB shootdown request sent to the VCPU formerly executing on the paused core.

The other four workloads, however, have much more frequent interaction among cores, causing rapid degeneration of the entire system's forward progress. For Apache and Zeus, nearly half of the VCPUs in the system stop making forward progress within 1ms of the fault. For the three commercial workloads in this graph, all VCPUs stop making forward within 3–11ms. The fault-free cores are simply executing OS spin loops waiting for either cross calls to complete, or locks held by the faulting processor [46]. While not

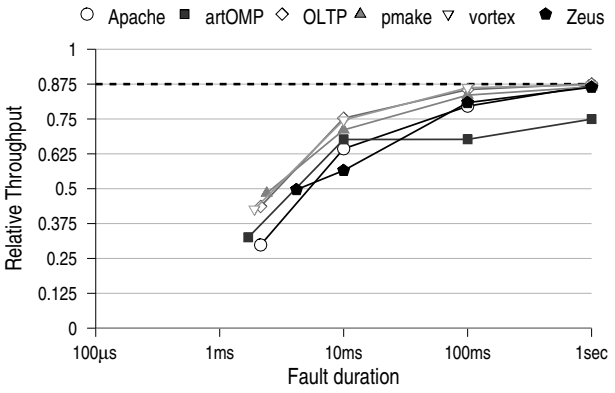

Figure 4. Throughput of OS Reconfiguration During a Fault

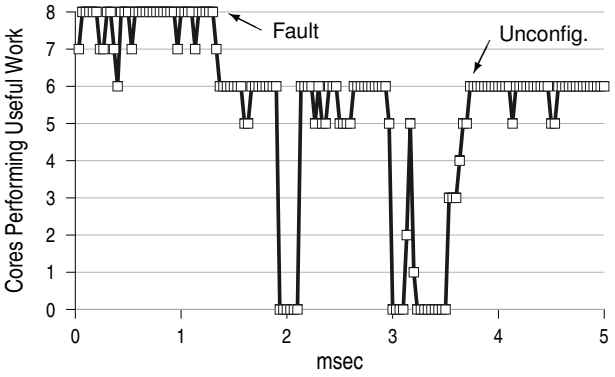

Figure 5. OS Reconfiguration of Zeus

Apache	artOMP	OLTP	pmake	vortex	Zeus
2.12	1.69	2.12	2.40	1.89	4.09

Table 2. Reconfiguration Latency (ms)

shown in the graph, all cores will eventually resume useful work after the paused core is re-enabled, provided the paused interval is short enough that the OS kernel does not panic ($\sim$1 second for Solaris 9).

Despite its simplicity, Figures 2 and 3 show that the cascading livelock suffered by many workloads makes pausing execution unattractive for long faults. On the other hand, for short ($<$1ms) periods, this technique may be appropriate in some environments.

OS Reconfiguration To determine the performance of OS Reconfiguration, we again inject faults of various durations in our simulation. For these experiments, we send an interrupt to the OS to *unconfigure* the VCPU that was running on the core sustaining the fault, as described in Section 4.

During the longer 100ms and 1sec fault durations, the cost of OS reconfiguration begins to amortize, and the throughput of all the workloads approaches the expected value of one less core compared to the baseline. For the shorter intervals, however, the cost of reconfiguration is not amortized — the loss in throughput is 2–6 times the loss expected from a single disabled core.

The time required for reconfiguration to complete is shown in Table 2. During this time, the OS requires all eight VCPUs to continue to execute code, by either using a fall-back adaptation mechanism, or by continuing to execute code on the faulty core itself. In addition, this latency also represents the minimum length of time that overheads from reconfiguration will be incurred, even if the

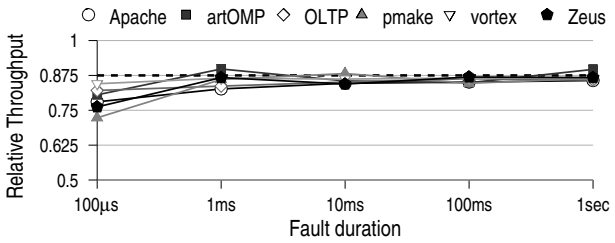

Figure 6. Throughput of Spare Cores

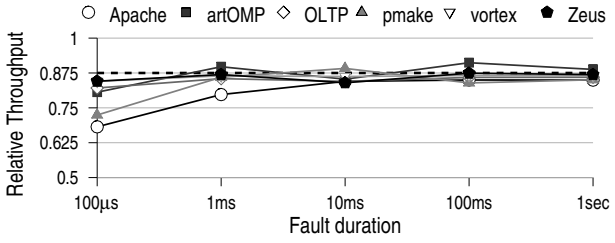

Figure 7. Throughput of an Overcommitted System

suspended core is re-enabled in the meantime (since reconfiguration cannot simply be *stopped* once in progress). The first point for each benchmark is thus placed on the x-axis (and measured against the baseline) at point in time that the VCPU is finally disabled.

Similar to Figure 3, Figure 5 explains this data by measuring useful work during various intervals. At 1.3ms (label 'Fault'), both the VCPU executing on the faulty core (Solaris's CPU4) and the recipient of the interrupt (Solaris's CPU3), stop committing user instructions. At 3.6ms (label 'Unconfig.'), CPU4 is finally unconfigured and enters a PROM idle loop. All processors in the system are quiesced twice to avoid deadlock arising from outstanding cross calls, according to comments in the source code. Note that the latencies in Table 2 are an average — this trace took only 2.3ms.

Spare Cores Spare cores can provide throughput during a fault that matches the fault-free throughput (which also uses one less core than the baseline). Figure 6 demonstrates this fact. For the shortest fault, 100μs, the 10k cycles we assume for recovering from the fault introduces some overhead. Likewise, the process of transferring VCPU state and then incurring misses on all cached data causes additional initial overhead. For all the longer durations, however, there is practically no loss in throughput compared to the best expectation. *artOMP* appears to incur sub-linear slowdown for certain runs. This is an artifact of our methodology for simulating spare cores (see Section 4): a VCPU in the baseline system enters a spin loop waiting for all other VCPUs to acknowledge one of the aforementioned TLB shootdowns, causing our perfect spin detection to yield the core to a productive thread.

Overcommitted System Figure 7 demonstrates the high performance of the overcommitted system. Similar to using spare cores, this technique incurs some overhead for the shortest faults due to the recovery time and cache misses. However, this overhead is small and is quickly amortized for longer fault rates.

Using an overcommitted system with spin detection during periods of intermittent faults yields throughput similar to using spare cores. Unlike spare cores, however, this technique retains the ability to utilize the entire machine during periods of fault-free execution, and can handle concurrent failures.

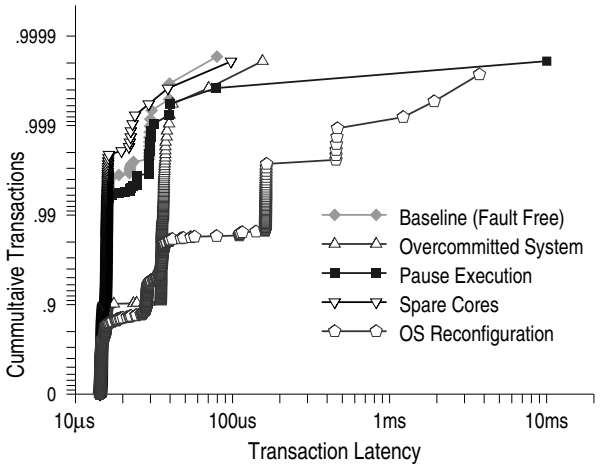

Figure 8. Microbenchmark Transaction Latencies

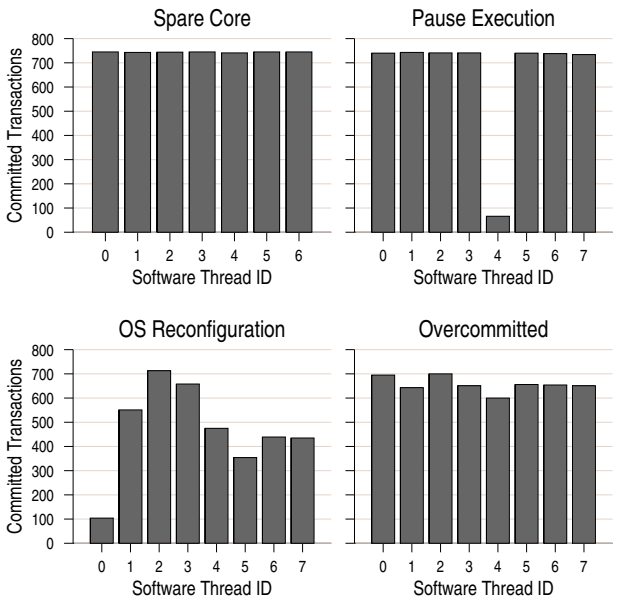

Figure 9. Microbenchmark Transactions from each SW Thread

5.2 Microbenchmarking Latency & Fairness

While throughput is important, other performance metrics are equally important for certain applications. For example, latency is critical for Multiplayer Online Games [15], or for telemetry applications, and fairness may be important for consolidated servers. Other metrics may be of interest as well. Ideally, we would use these target applications to measure transaction latency and fairness, but the complexity of building such workloads, combined with irregular or long transactions and the distorting effects of other software components, conspire to make such an evaluation difficult. Instead, we use a microbenchmark, described in Section 4, to understand the underlying behavior of our four adaptation techniques. We omit the data for different fault durations for brevity, but the results are easily extrapolated from the data for 10ms.

5.2.1 Latency

Figure 8 shows the cumulative distribution of transaction latencies from each software thread for our microbenchmark. Both axes are logarithmic to highlight transactions that deviate from the common case.

In the baseline, fault-free system, we see that 99.5% of transactions take 16μs or less, while several transactions take 40–100μs. We see very similar data when using a spare core, and when pausing execution, except that one transaction, the one on the paused core, takes over 10ms. Note that our microbenchmark, dominated by user code with no communication, represents the best case for pausing execution. With OS reconfiguration, many transactions are delayed by 100μs–1ms, while the OS quiesces all VCPUs. Because the OS migrates threads off the faulty core, no transactions are delayed as long as the 10ms fault, but many outliers remain.

When using an overcommitted system, the frequency with which VCPU context switching occurs can impact latency-sensitive applications. However, this frequency is configurable in firmware, and can be increased if necessary for a small increase in switching overhead. We have tuned our simulated firmware to perform a VCPU context-switch at least every 20μs, and thus observe that 12% of transactions take approximately 20μs longer than the baseline (since two VCPUs are vying for the same core). No outliers are delayed by more than 20μs longer than the baseline.

5.2.2 Fairness

To measure fairness, we examine the total number of transactions committed by each software thread. In Figure 9, we observe that the system with a spare core commits a nearly equal number of transactions per thread, and thus provides similar fairness as the baseline (not shown). This result assumes that the application software can be easily partitioned seven ways, which is not the case for many scientific applications. Note that the the graph for spare cores only has seven bars, while the others have eight.

	Base	Spare	Pause	OS	OverC
F.S. [11] ↑	1.00	1.00	0.44	0.49	0.94
ΣM_0 [24] ↓	0.92	1.00	5.47	7.14	1.17

Table 3. Fairness of Metrics for Different Techniques

We observe that the overcommitted system is able to provide conceptually similar fairness as spare cores and the baseline, even during the failure of one core. On the other hand, pausing execution causes one thread to be significantly impeded by the fault. Since the OS is still scheduling software threads among all eight VCPUs, one application thread is starved when pausing.

Due to the overhead of using at least one VCPU to orchestrate reconfiguration, and the quiescing of all VCPUs, OS reconfiguration cannot maintain fairness among software threads during the 10ms interval we simulate. We would expect that for longer fault durations, the OS might fare better.

To quantify the degree of fairness, we examine both the *fair speedup* (F.S.) metric used by Chang, et al. [11], and the ΣM_0 metric from Kim, et al. [24]. For fair speedup, we take the harmonic mean of the speedup between each software thread and the most productive thread in that experiment. ΣM_0 is derived from the sum of M_0 across all pairs of threads i, j, where $M_0^{ij} = \|X_i - X_j\|$, $X_i = \frac{Trans_i}{Trans_p}$, and p is the most productive thread.

For fair speedup, higher is better, and for ΣM_0, lower is better. These metrics are shown in Table 3. For both metrics, the baseline and spare cores are very close, while the overcommitted system is only slightly worse than both of them. Pausing and OS reconfiguration are significantly worse. Though the metrics differ in how much they penalize the OS and pausing schemes, both clearly show that these two techniques are inferior in terms of fairness.

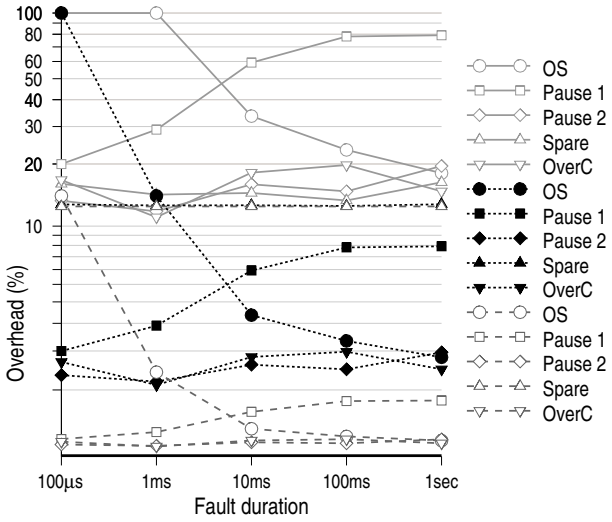

Figure 10. Overhead with Different Fault Duty-Cycles (Analytic Model) *Solid Gray Lines: 10%, Dark Dashed Lines: 1%, Light Dashed Lines: 0.1%*

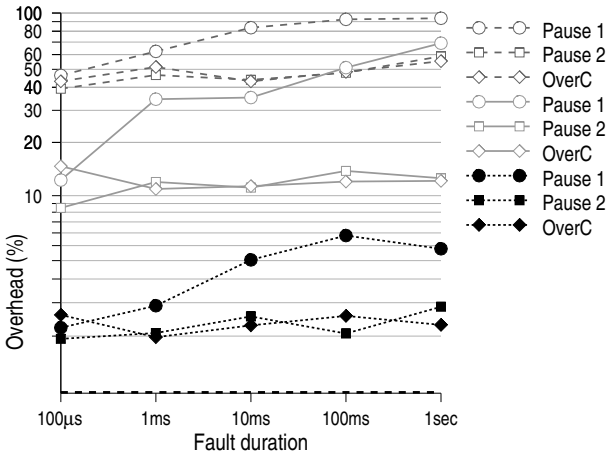

Figure 11. Overhead with Different Fault Duty-Cycles (Execution Driven Simulation) *Light Dashed Lines: 50%, Solid Gray Lines: 10%, Dark Dashed Lines: 1%*

5.3 Overall Impact of Different Fault Rates

Thus far we have examined throughput during faults without considering intervening periods of fault-free execution. Now we look at the overall impact of the four techniques across a range of fault durations and frequencies.

Using an analytic model, we extrapolate the throughput data from Section 5.1 to determine the overhead at various fault rates. We use an analytic model to examine overheads in a more controlled environment, because we cannot perform an execution-driven simulation of either OS reconfiguration due to limitations in Simics, or spare cores due to its inability to handle more concurrent faults than spares (see Section 4). We also present data for execution-driven simulations using the other two techniques to validate the model and to explore multiple concurrent failures.

5.3.1 Analytic Model

In our simple analytic model, we first average the overhead (1 - throughput) from all six benchmarks. We break the pause scheme into two groups, *Pause 1*, containing the commercial workloads (Apache, Zeus, OLTP, and pmake) for which pausing works poorly, and *Pause 2*, containing vortex and artOMP, for which pausing works well. Then, we factor in the expected fraction of time these techniques are employed during runs with various fault rates. For simplicity, we assume no concurrent faults.

In Figure 10, each line in the graph keeps the *duty-cycle* constant, i.e., the fraction of the time each core is experiencing a fault. Thus, 100μs faults with a duty cycle of 1% are occuring, on average, every 10ms, and 1sec faults are occurring, on average every 100sec. The solid gray lines near the top of the graph represent a duty cycle of 10%. The middle set of dark dashed lines represent a duty cycle of 1%, and the lower, lighter dashed lines represent 0.1%. Both axis are logarithmic. Because we assume no concurrent faults, space cores incurs $\sim$12.5% overhead for 0.1–1% duty cycle, and slightly more for 10%.

In all experiments, we see that the group *Pause 2*, as well as the overcommitted scheme, incur overheads from 1–2 times the duty cycle. The same is true for the group *Pause 1* for 100μs faults, and for OS reconfiguration for 1sec faults. However, for longer faults, we observe overheads of approximately eight times the duty cycle for *Pause 2*, since a fault on each core affects the other eight as well. Similarly for OS reconfiguration, not only does a fault on one core affect the others, but the latency of reconfiguration causes overheads 2–3 orders of magnitude larger that the duty cycle for the shortest faults.

All techniques are expected to incur low overheads when fault rates are low, but even when fault rates create a duty cycle of 0.1%, care must be taken when invoking the pause or OS reconfiguration techniques.

5.3.2 Execution-Driven Simulation

The simple analytic model in the previous section was unable to handle multiple concurrent failures, which is necessary in order to experiment with higher fault duty-cycles. In this section, we present results of execution-driven simulation using randomly generated periods of intermittent faults.

For longer faults, multiple concurrent failures actually benefit the pause scheme in comparison to the duty cycle, since other cores that might be affected by pausing one have some probability of already being paused themselves. This is evidenced in Figure 11 by the 100ms duration on the 10% duty-cycle line: the *Pause 1* incurs a 55% overhead with simulation, but a projected 80% overhead from our model.

Using an overcommitted system maintains the overall impact at a level commensurate with the duty-cycle for all applications in all experiments. It is also nearly the same as the *Pause 2* group (i.e., workloads which have little communication). In summary, using an overcommitted system yields low overhead, even when half of the cores, on average, are faulty. The same is not true for any of the other schemes.

5.4 Future Multicores

Based on what we can assume about future multicores, we believe that the qualitative results of our experiments will generally hold. Future technologies will allow room for many more than eight cores, and this will undoubtedly have an impact on techniques for adapting to intermittent faults. If applications are partitioned so that they each use no more cores that they do in our simulations, we would expect the results for pausing execution to be similar. However, this technique could be devastating if a single application, with occasional communication, is using all cores of the chip. As long as all the cores are under the control of a single OS, or single hypervisor, the system software may still have to quiesce all

	Quantitative Goals				Qualitative Goals		Appropriate Timescales
	Fairness	Latency	Throughput	No-Fault Cost	Complexity	Concurrent	
Pause Exec.	X	X	X	√	Low	√	≤1ms
Spare Cores	√	√	√	X	Med.	X	100μs–1sec+
OS Reconfig.	X	X	X	√	High	X	≥100ms
Overcommitted	√	√	√	√	Med.	√	100μs–1sec+

Table 4. Results Summary

cores to prevent deadlock, increasing the latency and overheads of software reconfiguration.

Using spare cores becomes more viable as fault rates increase and the relative granularity of spares decreases. However, this technique still cannot easily adapt to long or short-term changes in the number of concurrent faults. For example, when using a laptop on an airplane, or when one section of a data center becomes too hot, fault rates may increase, requiring more spares. At other times, few if any spares may be necessary. Setting the number of spares too high introduces overhead, and setting it too low increases the probability of observing more concurrent faults than spares. An overcommitted system, on the other hand, has a distinct advantage since it can dynamically adapt to these changes.

6. Related Work

Intermittent faults Many circuit-level techniques for tolerating intermittent faults have been proposed [4, 19, 20, 31], but they are generally applicable only to individual components. Consequently, they are likely to be useful for reducing the frequency, but not eliminating, intermittent faults. Similarly, thermal management techniques (e.g., [32, 38]) can be used to reduce the frequency of faults by managing thermal variations. However, for future processors, avoiding intermittent faults with these techniques will require them to be overly conservative, thus providing low performance.

Reconfiguring after Device-level Faults Several methods have been presented to continue use of a core despite permanent faults. These techniques involve fine-grained diagnosis and reconfiguration of a core's components [9,37], or attempt to match a program's requirements and a core's capabilities, such as Core Salvage [21]. We believe that the ability to suspend execution on a core in order to perform diagnosis and reconfiguration would likely be a simplifying addition to these techniques.

Fault Tolerance in Distributed Systems Much distributed systems research has addressed fault tolerance for clusters of computers, e.g., [3, 5, 14, 17, 23]. For most of this research, the unit of failure is an entire machine, including the cpu(s), memory, and system software. Such course-grained units are not applicable to systems comprised of only a few, or even one, multicore chip.

In addition, the comparatively short timescales of device-level intermittent faults render these software-based adaptation techniques ineffective because they cannot adapt quickly enough (see Section 3.1). For example, if certain cores on a chip observe intermittent faults every few seconds, software techniques will, by necessity, consider the entire chip to be permanently faulty.

Chameleon [23] provides a reliable software-based fault tolerant system. They use the term *Adaptive Fault Tolerance* to describe a system that is flexible to the dynamic demands of applications, but not necessarily to the dynamic conditions of the hardware.

7. Conclusions

As technology continues to scale, the effects of intermittent faults will become important considerations in multicore design. Although complex reliability techniques may tolerate many intermittent faults without affecting the rest of the system, we believe these approaches will require, or be greatly simplified by, the ability to temporarily suspend computation on a core during bursts of such faults.

We examine the system-level implications of thee obvious mechanisms for adapting to the temporary loss of one or more cores, and show that all three have serious deficiencies as summarized in Table 4. To remedy these drawbacks, we propose a fourth technique: using a thin hardware/firmware layer to manage an *overcommitted system* — one where the OS is configured to use more virtual processors than the number of currently available physical cores. Utilizing an overcommitted system is the only mechanism to achieve high marks on all of the performance metrics across a range of timescales, gracefully handle multiple concurrent failures, and involve only moderate complexity.

By eliminating the system-level concerns through our proposed overcommitted system, we believe researchers will find the ability to suspend execution on a core to be a useful tool — both to simplify the design and improve the coverage of reliable chips, and for other uses that have yet to be discovered. Furthermore, we motivate our work using intermittent faults, yet a variety of factors will cause the resource configurations of future multicores to dynamically and frequently vary. We believe that the flexibility of techniques such as our proposed overcommitted system will allow architects and system designers to continue exploring the opportunities and challenges of this frequent resource variation.

Acknowledgments

We thank Jichaun Chang and Matthew Allen for many helpful discussions, and also Luke Yen and the anonymous reviewers for their comments. This work is supported in part by National Science Foundation (NSF) grants CCF-0702313 and CNS-0551401, funds from the John P. Morgridge Chair in Computer Sciences and the University of Wisconsin Graduate School. Sohi has a significant financial interest in Sun Microsystems. The views expressed herein are not necessarily those of the NSF, Sun Microsystems or the University of Wisconsin.

References

[1] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Prog.*, Dec 2005.

[2] W. Armstrong et al. Advanced virtualization capabilities of POWER5 systems. *IBM Journal and Research and Development*, 49(4/5), 2005.

[3] D. Bernick et al. Nonstop advanced architecture. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, 2005.

[4] D. M. Blough, F. J. Kurdahi, and S. Y. Ohm. High-level synthesis of recoverable VLSI microarchitectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(4):401–410, 1999.

[5] D. M. Blough, G. F. Sullivan, and G. M. Masson. Intermittent fault diagnosis in multiprocessor systems. *IEEE Transactions on Computers*, 41(11):1430–1441, 1992.

[6] S. Borkar. Microarchitecture and design challenges for gigascale integration: Keynote. In *Proceedings of the 37th Annual International Symposium on Microarchitecture (MICRO)*, 2004.

[7] S. Borkar, T. Karnik, and V. De. Design and reliability challenges in nanometer technologies. In *Proceedings of the 41th Annual Conference on Design Automation*, 2004.

[8] S. Borkar, T. Karnik, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *Proceedings of the 40th Annual Conference on Design Automation*, 2003.

[9] F. A. Bower, D. J. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proceedings of the 38th Annual International Symposium on Microarchitecture (MICRO)*, 2005.

[10] K. Bowman, S. Duvall, and J. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *IEEE Journal of Solid-State Circuits*, 37(2):183–190, Feb 2002.

[11] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st Annual International Conference on Supercomputing*, 2007.

[12] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23(4):14–19, 2003.

[13] C. Constantinescu. Intermittent faults in VLSI circuits. In *Proceedings of the IEEE Workshop on Silicon Errors in Logic - System Effects*, 2007.

[14] O. Contant, S. Lafortune, and D. Teneketzis. Diagnosis of intermittent faults. *Discrete Event Dynamic Systems*, 14(2):171–202, 2004.

[15] G. Deen, M. Hammer, J. Bethencourt, I. Eiron, J. Thomas, and J. Kaufman. Running Quake II on a grid. *IBM Journal and Research and Development*, 45(1), 2006.

[16] D. Ernst et al. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO)*, 2003.

[17] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular Disco: Resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 18(3):229–262, Aug 2000.

[18] S. H. Gunther, F. Binns, D. M. Carmean, and J. C. Hall. Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal*, Q1, 2001.

[19] S. N. Hamilton and A. Orailoglu. Transient and intermittent fault recovery without rollback. In *Proceedings of the 13th International Symposium on Defect and Fault-Tolerance in VLSI Systems*, 1998.

[20] A. A. Ismaeel and R. Bhatnagar. Test for detection & location of intermittent faults in combinational circuits. *IEEE Transactions on Reliability*, 46(2):269–274, Jun 1997.

[21] R. Joseph. Exploring core salvage techniques for multi-core architectures. In *Proceedings of the Workshop on High Performance Computing Reliability Issues*, 2006.

[22] R. Joseph, D. Brooks, and M. Martonosi. Control techniques to eliminate voltage emergencies in high performance processors. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA)*, 2003.

[23] Z. T. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant. Chameleon: A software infrastructure for adaptive fault tolerance. *IEEE Transactions on Parallel and Distrubuted Systems*, 10(6):560–579, 1999.

[24] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2004.

[25] C. LaFrieda, E. Ípek, J. F. Martínez, and R. Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *Proceedings of the 2007 International Conference on Dependable Systems and Networks*, 2007.

[26] T. Li, A. R. Lebeck, and D. J. Sorin. Spin detection hardware for improved management of multithreaded systems. *IEEE Transactions on Parallel and Distrubuted Systems*, 17(6):508–521, 2006.

[27] X. Liang and D. Brooks. Mitigating the impact of process variations on processor register files and execution units. In *Proceedings of the 39th Annual International Symposium on Microarchitecture (MICRO)*, 2006.

[28] T. Litt. Method and apparatus for CPU failure recovery in symmetric multi-processing systems. U.S. Patent 5,815,651, Sep 1998.

[29] P. Magnusson et al. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb 2002.

[30] S. Mitra, M. Zhang, N. S. amd T. M. Mak, and K. Kim. Soft error resilient system design through error correction. In *Proceedings of the Very Large Scale Integration*, January 2006.

[31] T. Nanya and H. A. Goosen. The byzantine hardware fault model. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(11):1226–1231, Nov 1989.

[32] M. D. Powell, M. Gomaa, and T. N. Vijaykumar. Heat-and-run: leveraging SMT and CMP to manage power density through the operating system. In *Proceedings of the 11th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[33] M. D. Powell and T. N. Vijaykumar. Exploiting resonant behavior to reduce inductive noise. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[34] Semiconductor Industry Association. International technology roadmap for semiconductors: Executive summary, 2005.

[35] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003.

[36] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, 2002.

[37] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra low-cost defect protection for microprocessor pipelines. In *Proceedings of the 12th International conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[38] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA)*, 2003.

[39] T. J. Slegel et al. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

[40] J. C. Smolens, B. T. Gold, J. C. Hoe, B. Falsafi, and K. Mai. Detecting emerging wearout faults. In *Proceedings of the IEEE Workshop on Silicon Errors in Logic - System Effects*, 2007.

[41] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA)*, 2002.

[42] Sun Microsystems. Sun fire high-end and midrange systems dynamic reconfiguration user's guide. http://docs.sun.com/app/docs/doc/819-1501. Viewed 12/19/2007.

[43] J. W. Tschanz, S. G. Narendra, Y. Ye, B. A. Bloechel, S. Borkar, and V. De. Dynamic sleep transistor and body bias for active leakage power control of microprocessors. *IEEE Journal of Solid-State Circuits*, 38(11), 2003.

[44] R. Uhlig et al. Intel virtualization technology. *Computer*, 38(5), 2005.

[45] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proceedings of the 3rd Virtual Machine Research and Technology Symposium*, 2004.

[46] P. M. Wells, K. Chakraborty, and G. S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *Proceedings of the 15th Annual International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2006.

[47] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005.