Adapting to Intermittent Faults in Multicore Systems

Philip M. Wells, Koushik Chakraborty, and Gurindar S. Sohi
Computer Sciences Department
University of Wisconsin, Madison
{pwells, kchak, sohi}@cs.wisc.edu

ABSTRACT

Future multicore processors will become more susceptible to a variety of hardware failures. In particular, intermittent faults, caused in part by manufacturing process variation or in-progress wear-out, can cause bursts of frequent faults that last from several cycles to several seconds or more. Cost-effective reliability to tolerate intermittent faults will likely require, or be greatly simplified by, the ability to temporarily suspend execution on a core during periods of frequent intermittent faults. We investigate three existing techniques for adapting to the dynamically changing resource availability caused by such core suspension, and demonstrate their different system-level implications.

We show that system software reconfiguration has very high overhead for short intermittent faults, that temporarily pausing the execution of 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 current 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 durations with low overhead, without involving system software, and without requiring spare cores.

1. Introduction

The components of future multicore processors will become unreliable as technology scales, because individual devices are increasingly susceptible to a variety of hardware faults [7, 9, 11, 14, 38, 40]. In particular, technology experts warn about the pending increase in *intermittent* faults — faults which occur unpredictably for a period of time, commonly due to process variation or in-progress wear-out, combined with voltage and temperature fluctuations [7, 9, 14, 15]. 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 fault tolerance 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 [5,18,21,22,30,34, 35, 41, 44]. Such techniques, however, have weaknesses along one or more axis of *fault coverage*; *time, power, and area overhead*; 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. It is increasingly unlikely (and perhaps not even desirable) that techniques to tolerate faults could do so in a manner that is completely transparent to system and application software.

Tolerating intermittent faults presents further challenges for designers. Unlike transient faults, which disappear immediately, intermittent faults occur in bursts which can last from several cycles to several seconds or more. While it may be possible for complex hardware and/or software fault tolerance schemes to deal with a variety of intermittent faults, and do so transparently to software, we believe that such schemes will be neither practical nor desirable. Rather, various reliability enhancement mechanisms will require, or be greatly simplified by, the ability to temporarily suspend the execution of code on a core that is sustaining intermittent faults. We believe that *core suspension* will be an effective policy for 1) reducing several of the factors contributing to the faults in the first place, 2) reducing the number of faults that must be corrected by other components, and 3) aiding in the diagnosis of permanent circuit damage.

Naively suspending a processing core is not common practice because it is not transparent to software and can have serious system-level implications. Fortunately, multicore processors provide unique opportunities to enable several techniques for adapting to the temporary loss of one or more cores. Through the use of such techniques, the system can *adapt* to the dynamically changing resource availability created by intermittent faults, possibly without a drastic performance (or other) impact on the system. This paper is concerned with understanding the system-level implications and effectiveness of these techniques. We present a qualitative and quantitative comparison of three existing adaptation techniques, and demonstrate their different system-level effects. The techniques are 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. To remedy several drawbacks of first three, we propose a fourth technique: using a thin hardware/firmware layer to manage an *overcommitted* system — a term used by Wells, et al., to denote a system where the OS is configured to use more virtual processors than the number of physical cores [50].

The primary objective of this paper is to analyze these four techniques, and help determine how frequently they can be invoked without unduly affecting system and application-level software. Our experiments indicate that utilizing an overcommitted system is the only technique to achieve high marks on all of the performance metrics across a range of timescales, gracefully handle multiple concurrent failures and high fault rates, and involve only moderate complexity. But we also show that each of the other three adaptation techniques may also be adequate in certain circumstances.

Because they can now do so without fear of negative system-level consequences, we believe that researchers should thus consider how the ability to suspend execution on a particular core might be helpful, whether to simplify the design and improve coverage of reliability mechanisms, or for other undiscovered uses.

Before exploring these and other results in more detail (Section 5), we discuss the causes of intermittent faults and the derivations of our major assumptions (Section 2), explore the qualitative properties of the four adaptation techniques (Section 3), and provide details of our experimental 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 [7, 9, 14, 15]. These variations can result in timing errors even when operating conditions are well within the specified "acceptable" noise margins.

Because intermittent faults are affected by a large number of factors, the duration of bursty faults can occur across a wide range of timescales. For example, voltage fluctuations are typically short-lived, on the order of several to hundreds of nanoseconds [9,24,37]. Temperature fluctuations alter a device's timing characteristics over millisecond to second time scales [36]. Different software phases, which can change on the order of 100msec to several seconds [39], can exercise different components of a core, activating different intermittent faults. Finally, as wear-out progresses over the course of days [44], it can cause intermittent faults to become frequent enough to be classified as permanent [14].

In this paper, we make three primary assumptions regarding intermittent faults. Below we examine the insights that lead us to believe these assumptions are reasonable.

1) Bursty "intermittent" faults will occur frequently.

While it is not clear what the exact rates of various faults will be 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 [8,9,38], but devices typically do not fail suddenly, they display intermittent behavior for a period of time beforehand [14, 15, 44]. Second, continued device scaling will result in increased PVT variations, increased cross-talk, and decreased noise margins [8, 9, 11, 14, 15, 38], all of which lead to increased susceptibility to intermittent timing faults. In this paper, we choose to examine fault rates in the range where they begin to impact system performance.

2) Practical circuits cannot mask all intermittent faults.

While many techniques for tolerating various faults have been proposed [5, 18, 21, 22, 30, 34, 35, 41, 44], 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 [18] 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, suspending the use of a core *will* cause temperature and voltages to stabilize, two major factors contributing to the occurrence of intermittent faults.

3b) ... reduces faults.

All reliability techniques have a certain probability of protecting against faults, which may be, for example, 90% or 99.999%. Certain events (e.g., multiple concurrent faults, or faults affecting critical structures) will still be permitted to manifest as an unprotected error. Suspending the use of a core when a burst of faults begins, or is expected to occur, can dramatically reduce the number of faults that must be protected by other techniques. If the number of faults requiring protection is reduced, then the number of events that remain unprotected will also be reduced, thus improving the overall reliability of the system.

Current high-availability systems already do something similar by having service technicians replacing chips when intermittent faults begin to occur [14]. 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 undesireable.

3c) ... is likely to be useful for other purposes.

Several proposals have appeared that call for fine-grained reconfiguration of a core's components (e.g., [10, 41]), or match a program's requirements to a particular core's degraded capabilities, (e.g., [23]). We believe that the ability to suspend execution on a core, without significantly impacting the rest of the system, makes these techniques more feasible. Additional uses for suspending a core will likely be discovered as architects reconsider the plausibility of such a policy.

3. Adapting to Intermittent Faults

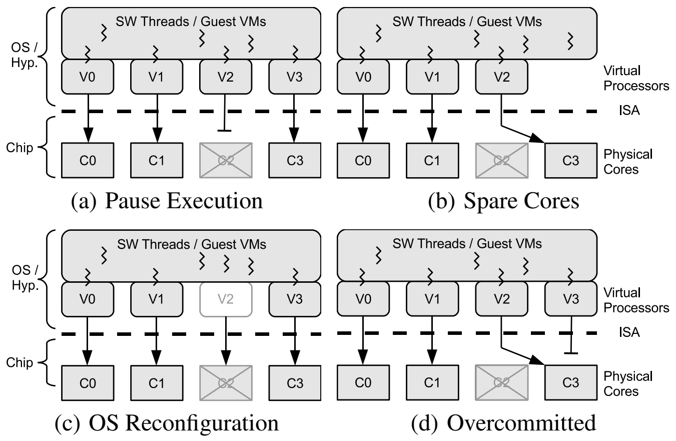
Naively suspending a processing core is not common practice because it is not transparent to software and can have serious system-level implications. Fortunately, multicore processors provide unique opportunities, including inherent redundancy, low on-chip latency, and high bandwidth, which enable several 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 of each of the first three. In our discussions, we pay particular attention to the system-level implications of these techniques. 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 as simply *cores*. We refer to the software-visible processing units as *virtual processors* or *VCPUs*. In many cases, the two are equivalent. However, in certain circumstances, the hardware/firmware may 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.

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 [20]. 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



(a) Pause Execution (b) Spare Cores (c) OS Reconfiguration (d) Overcommitted

Figure 1: Core Suspension Techniques

paused core. If communication is present, however, pausing one core can cause a cascading effect, livelocking other cores.

This technique is not *fair*, because any 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 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 an eight-core chip, only seven might be exposed 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 [47]. 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) [4, 43].

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, using spare cores cannot tolerate more concurrent failures that the number of spares without an additional fall-back mechanism.

Technique 3: OS/Hypervisor Reconfiguration A third possible technique is to ask the operating system (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. Though software intrusive, some current OSs (such as Solaris) and hypervisors (such as those that run on the IBM zSeries) already contain this functionality [2, 46].

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/hypervisor 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 system software requires the faulty core, and all other cores, to operate correctly until reconfiguration is complete [31, 46]. 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: 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. Unlike the overcommitted system used in [50], however, we propose a simple hardware/firmware layer that is hidden under the ISA, and thus operates beneath both the OS and a traditional hypervisor such as VMWare or IBM's Power5 Hypervisor. For simplicity, we typically refer to the lowest software layer as the *OS*, though that could be replaced with *hypervisor* throughout with no loss of generality.

In an overcommitted system, two (or more) OS-visible virtual processors (VCPUs) must share a single physical core. Figure 1(d) shows this technique, with virtual processors V2 and V3 sharing core C3. V2 is currently executing, while V3 is paused, but they can frequently switch to avoid the issues associated with pausing. The virtual processors 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 [29, 50], where a virtual processor that is not currently running could be holding a kernel lock or be the recipient of a CPU cross-call. Spin detection preempts requesters spinning on the lock, or initiators waiting for acknowledgment of the cross-call, in favor of virtual processors that are performing useful work. Spin detection is *not* required for correctness, as long as the hardware/firmware periodically forces a context 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, 48].

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.

While spin detection hardware is helpful with unmodified Solaris, 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 requirement [2,49].

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, chip manufacturers can 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. Such a model has several advantages 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. 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., [31]), it is clearly infeasible for others. Instead, we assume that the fault recovery mechanism periodically creates checkpoints, similar to [41], [27], or [45]. The checkpoints are stored into the cache every 10k cycles, and on I/O, and are consistent across the on-chip cores [27,45].

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. Depending upon the exact detection mechanisms, however, it may be possible to separate one faulty core from a logical pair and allow the non-faulty core to form a logical pair with another physical core (e.g. [27]).

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 [32], 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, for longer experiments, an in-order core to reduce simulation time. 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 [50].

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 For the experiments in Section 3.1, we instruct Solaris to *unconfigure* one of its eight virtual processors, CPU4. To perform this task in our simulations, we send 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 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 *vortex* from SpecINT2000 running ref 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 [3]. We do not use any think time in the Surge client. *pmake* is a parallel compile of PostgreSQL using GNU make and the Sun Forte Developer 7 C compiler. We do not include serial phases. Due to workload variability, we simulate multiple runs and report average results, though we omit confidence intervals for readability. We use a microbenchmark in Section 5.3, which consists of one thread per processor, which 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 [51].

5. Experimental Results

Intermittent faults requiring the suspension of a core can be caused by a variety of factors, and can last for a range of durations, and can affect applications in different ways. We first discuss our experiments, and then quantify the strengths and weaknesses of the four techniques along several axes.

5.1 Experiments

Measuring Throughput For throughput experiments in Section 5.2, one core experiences a detected fault at the beginning of execution; simulations are then run for a range of times from 100μsec to 1 second. We use the in-order processor for 1sec data points and the OOO for the rest.

Measuring Latency and Fairness For both the latency and fairness experiments in Section 5.3, we use our microbenchmark and simulate a single 10msec fault beginning at 100μsec of simulation, and then run for 11msec. The spare core experiments have seven threads and seven processors, with eight threads and eight processors for the rest. We use the OOO processor.

Measuring Overhead To determine overheads, including periods of fault-free execution, 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 run simulations for 1sec and use the in-order processor.

We properly 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 be *already* 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 be computationally intractable for these experiments.

Tracing Faults In order to better understand the results of the other experiments, Figures 3 and 5 trace the number of cores performing *useful work* during a fault of 100msec. We define *useful work* for each processor as whether or not any user instructions were executed in each 100μs period, and sum this boolean value over all eight OS-visible processors. These plots both use the OOO processor.

5.2 Throughput During a Fault

In this section, we demonstrate the throughput of all four techniques *during* intermittent faults of various duration. We discuss each technique in turn below. A line is drawn at $\frac{7}{8}$ in throughput graphs to represent the expected slowdown of losing one core.

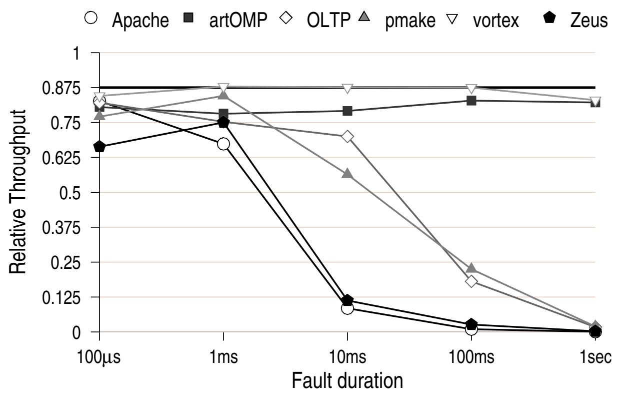


Figure 2: Throughput of Pausing Execution During Faults of Various Duration

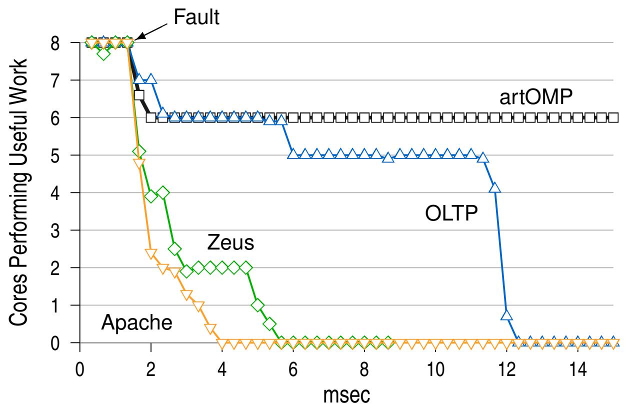


Figure 3: Cascading Livelock of Pause Scheme

Pause Execution Figure 2 shows the throughput of each benchmark when pausing execution for faults of various duration. For the shorter duration faults, 100μsec and 1msec, all workloads observe throughput that is within 25% of the expected case after losing one core. Across a range of fault durations, *vortex* continues to have throughput similar to the expected 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 1msec — even approaching *zero* throughput for 100msec and 1sec duration faults.

Figure 3 helps explain this loss of throughput for longer faults. This figure shows the first half of a trace of the number of cores performing useful work during every 0.3msec of a 20msec fault. For all workloads, the number of cores performing useful work immediately drops to seven (or lower) after the fault. For *artOMP*, a second core quickly stops performing work because it has blocked waiting on a lock held by the paused core. We do not move the VCPU executing on the faulty core elsewhere, so the lock is never released. Other VCPUs for *artOMP* remain unaffected well beyond the 50msec shown in the graph. We do not graph *pmake* or *vortex* for clarity.

The other 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 1msec. For the three commercial workloads in this graph, all VCPUs stop making forward within 7–32msec. The fault-free processors are simply executing OS spin loops waiting for either cross calls to complete,

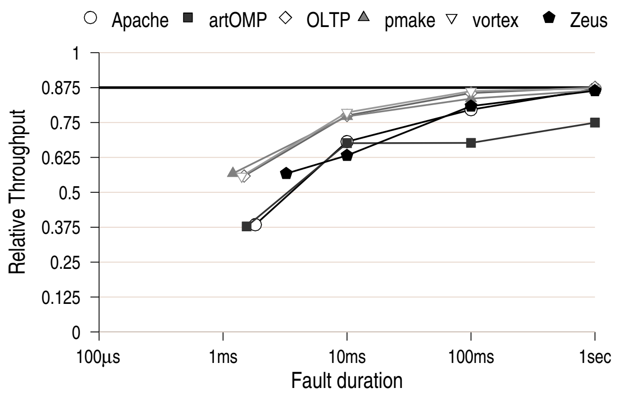


Figure 4: Throughput of OS Reconfiguration During a Fault

Apache	artOMP	OLTP	pmake	vortex	Zeus
1.81	1.55	1.50	1.19	1.41	3.22

Table 1: Reconfiguration Latency (msec)

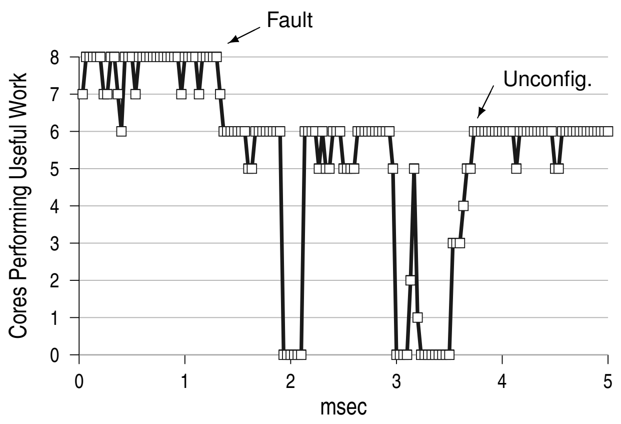


Figure 5: OS Reconfiguration of Zeus

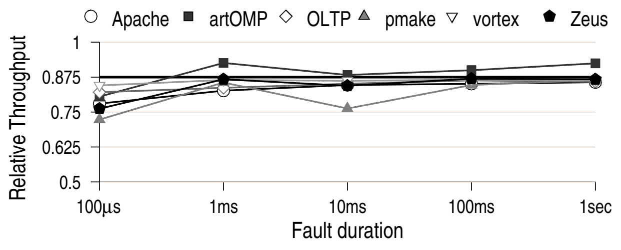


Figure 6: Throughput of Spare Cores

or locks to be unlocked by the faulting processor [50]. While not shown in the graph, all processors will eventually return to work after the paused core is resumed, provided the paused interval is short enough that the OS kernel doesn't panic ($\approx$1 second for Solaris 9).

Despite its simplicity, Figures 2 and 3 shows 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 simulate faults of various duration. For these experiments we send an interrupt to the OS to tell it to *unconfigure* its 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 be amortized, 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 1, and is interesting because this is the length of time that this technique requires a fall-back mechanism.This latency also represents the minimum length of time that overheads from reconfiguration will be incurred, even if the suspended core is reenabled in the meantime (since reconfiguration cannot simply be *stopped* once in progress). With this in mind, the first point for each benchmark is placed on the x-axis (and measured agains the baseline) at the location that the VCPU is finally disabled.

Again, we turn to our tracing experiments, and Figure 5, for help explaining this data. At 1.3msec (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. Note that all processors in the system are quiesced twice (to avoid possible deadlock arising from outstanding cross calls, according to comments in the source code). Also note that the 3.2ms latency the table of Figure 4 is an average — this particular trace took only 2.3ms.

Spare Cores Using spare cores can provide throughput during a fault that matches the fault-free throughput with one less core than the baseline configuration. The dashed lines in Figure 6 demonstrate the this fact. For the shortest fault, 100μsec, 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 expected case. *artOMP* appears to incur sub-linear slowdown for certain runs. This is an artifact of our methodology for simulating spare cores (see Section 4): one VCPU in the baseline system becomes idle for part of the simulation due to benchmark synchronization, causing our perfect spin detection to yield the core to a productive thread. The slowdown for *pmake* can be attributed to workload variability.

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 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, yet retains the ability to utilize the entire machine during periods of fault-free execution.

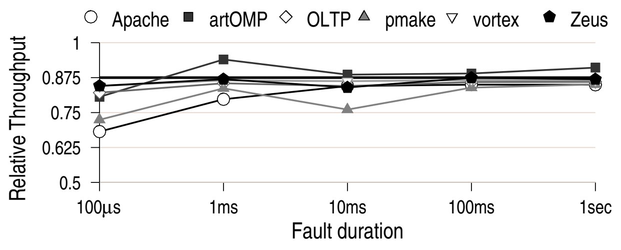


Figure 7: Throughput of an Overcommitted System

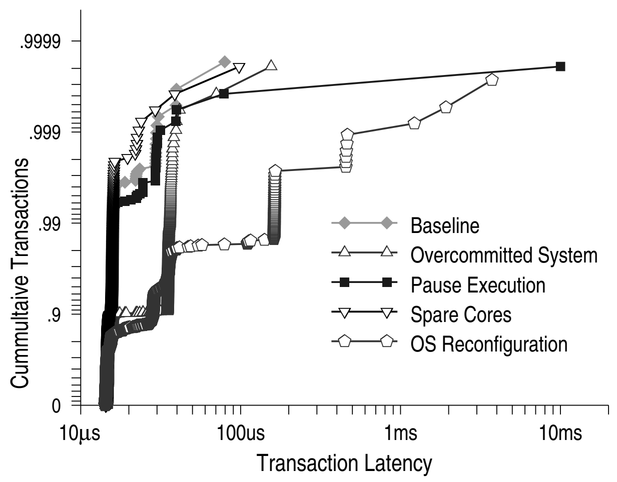


Figure 8: Microbenchmark Transaction Latencies

Base	Spare	Pause	OS	OverC
1.00	0.99	1.12	1.42	1.12

Table 2: Average Latency (Relative to Baseline)

5.3 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 [17], or for telemetry applications, and fairness may be important for consilidated 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 particularly 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 straightforward to extrapolate from the data for 10ms.

5.3.1 Latency

Figure 8 figure 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 the common case.

In the baseline, fault-free system, we see that 99.5% of transactions take 16μsec or less, while several transactions take 40–100μsec. 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 10msec. Note that our microbenchmark, dominated by user code with no communication, represents the best case for pausing execution. Average latencies are shown in Table 2, and we see that the spare cores is again similar to the baseline, since no transactions are ever *started* on the core that will sustain a fault. The outlier when pausing execution increases the average for that technique by 12%.

With OS reconfiguration, many transactions are delayed by 100μsec–1msec while the OS quiesces all processors in the system. But because the OS migrates threads off the faulty core, no transactions are delayed quite as long as the 10msec fault, but there are many outliers, and average latency 42% higher than the baseline.

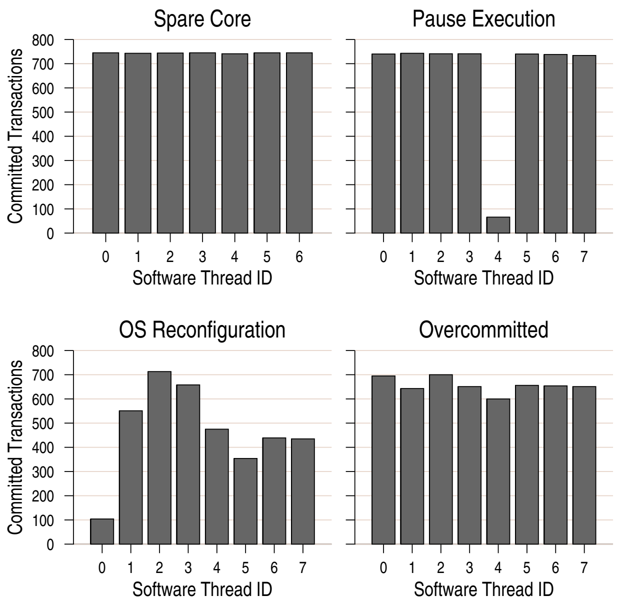


Figure 9: Committed Transactions from each Software Thread

	Base	Spare	Pause	OS	OverC
F.S. [13] ↑	1.00	1.00	0.44	0.49	0.94
ΣM_0 [26] ↓	0.92	1.00	5.47	7.14	1.17

Table 3: Fairness of Metrics for Different Techniques

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 20μsec, and thus observe that approximately 10% of transactions take approximately 20μsec longer than the baseline (since two VCPUs are vying for the same core). This yields an average latency equivalent to pausing, but unlike pausing or OS reconfiguration, there are no extreme outliers.

5.3.2 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 partitioned seven ways like our microbenchmark can. Note that the the graph for spare cores only has seven bars, while the others have eight.

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* has one thread which is 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. [13], and the ΣM_0 metric from Kim, et al. [26]. For fair speedup, we take the harmonic

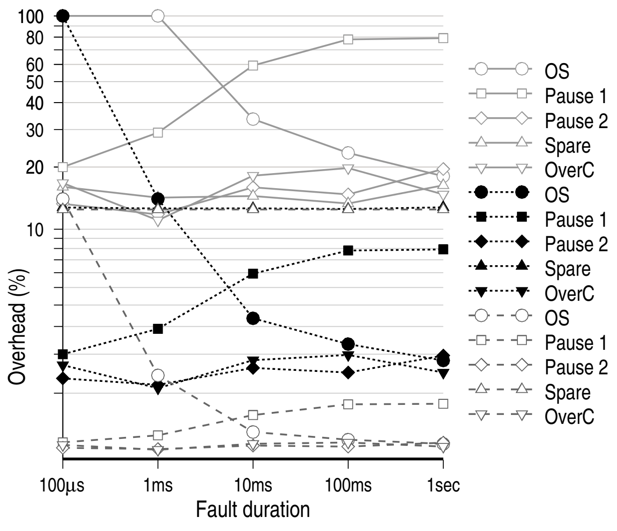


Figure 10: Overhead with Different Fault *Duty-Cycles* (Analytic Model) *Solid Gray Lines: 10%, Dark Dashed Lines: 1%, Light Dashed Lines: 0.1%*

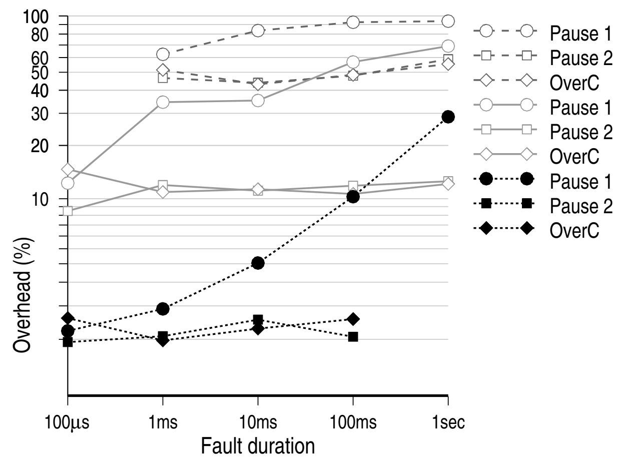


Figure 11: Overhead with Different Fault *Duty-Cycles* (Execution Driven Simulation) *Solid Gray Lines: 50%, Dark Dashed Lines: 10%, Light Dashed Lines: 1%*

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, they both clearly show that both are inferior in terms of fairness.

5.4 Overhead 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 overheads of the four techniques across a range of fault durations and frequencies.

Using an analytic model, we extrapolate the throughput data from Section 5.2 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 is 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 to back up the model, and to explore the case of multiple concurrent failures.

5.4.1 Analytic Model

In our simple analytic model, we first average the overhead (1 - throughput) from all 6 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 (an unlikely case for higher fault rates).

In Figure 10, each line in the graph holds the *Duty-Cycle* constant, i.e., the fraction of the time each core is experiencing a fault. Thus, 100μsec faults with a duty cycle of 1% are occuring, on average, every 10msec, and 1sec faults are occuring, on average every 100sec. The solid grey lines near the top of the graph represent a duty cycle of 10%. The roughly middle set of dark dashed lines represent a duty cycle of 1%, and the lower ligher dashed lines represent 0.1%. Both axis are logarithmic. Because we assume no concurrent faults, *Spare Cores* incurs $\sim$12.5% overhead in all experiments.

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 100μsec faults, and for *OS Reconfiguration* for 1sec faults. However, for longer faults, we observe overheads of approximately 8 times the duty cycle for *Pause 2*, since a fault on each core affects the other 8 as well. Similarly for OS reconfiguration, not only does a fault on one core affect the others, but the latency of reconfiguration creates overheads well beyond 100 times 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 fault duty cycles of 0.1%, care must be taken when invoking the Pause or OS reconfiguration techniques.

5.4.2 Execution-Driven

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 benefits the pause scheme in comparison to the duty cycle, since other cores that are likely to be affected by pausing one have some probability of being paused already 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 an projected 80% overhead from our model. On the other hand, for a duty cycle of 1%, where we do not expect concurrent failures, *Pause 1* is much worse than that expected by the model for 1sec faults.

Using an overcommitted system achieves overhead commensurate with the duty-cycle 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.

	Quantitative Goals				Qualitative Goals		Appropriate
	Fairness	Latency	Throughput	No-Fault Cost	Complexity	Concurrent	Timescales
Pause Exec.	X	X	$\surd$/X	$\surd$	Low	$\surd$	$\leq$1ms
Spare Cores	$\surd$	$\surd$	$\surd$	X	Med.	X	100μs–1sec+
OS Reconfig.	X	X	$\surd$/X	$\surd$	High	X	$\geq$100ms
Overcommitted	$\surd$	$\surd$	$\surd$	$\surd$	Med.	$\surd$	100μs–1sec+

Table 4: Results Summary

The same is not true for any of the other schemes, except when pausing certain applications.

5.5 Future Multicores

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. We examine chips with eight cores in part due to the practical limitations of full-system simulation, and in part because we do not know how chips with hundreds of cores will be used. Based on what we can assume, however, we believe that the results of our experiments will generally hold.

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

Using spare cores becomes more viable as fault rates increase and the granularity of spares decreases. However, it still cannot easily adapt to long or short-term changes is the number of concurrent faults. For example, when using a laptop on an airplane, or when one section of a datacenter becomes extra warm, 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 than that number of concurrent faults. An overcommitted system, on the other hand, had a distinct advantage since it can automatically adapt to these changes.

6. Related Work

Intermittent faults Many circuit-level techniques for tolerating intermittent faults have been proposed [5, 21, 22, 35], 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., [12, 36, 42]) 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 [10, 41], or attempt to match a program's requirements and a core's capabilities, such as Core Salvage [23]. 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., [4, 6, 16, 19, 25, 28, 33]. 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.

The comparatively short timescales of device-level intermittent faults render 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 [25] 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 multicore design considerations. 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 a core during periods of intermittent faults.

In this paper, we examine the system-level implications of four mechanisms for adapting to the loss of one or more cores. Table 4 summarizes these results. Although simply pausing execution is the most naive approach, it performs adequately for short duration faults, and is very simple. However, it can incur cascading livelock for faults over 1msec.

Setting aside one or more cores as *spares* avoids the problems of pausing, but has a high cost during fault-free execution, and cannot handle more concurrent faults the the number of spares. Expecting the Operating System (OS) to suspend its use of a core requires a long lead-time and has high overhead. Both techniques are hard to recommend for tolerating intermittent faults.

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.

Finally, we argue that, by eliminating the system-level concerns through our proposed overcommitted system, researchers will find the ability to suspend execution on a core to be a useful tool — both for simplifying the design and improving coverage of reliable chips, and for other uses that have yet to be discovered.

8. 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 J. Res. and Devel.*, 49(4/5), 2005.

[3] P. Barford and M. Crovella. Generating representative web workloads for network and server performance evaluation. In *1998 Conf. on Meas. & Model. of Comp. Sys.*, 1998.

[4] D. Bernick et al. Nonstop advanced architecture. In *Proc. of 2005 DSN*, 2005.

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

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

[7] S. Borkar. Microarchitecture and design challenges for gigascale integration: Keynote. In *Proc. of 37th MICRO*, 2004.

[8] S. Borkar, T. Karnik, and V. De. Design and reliability challenges in nanometer technologies. In *Proc. of 41th DAC*, 2004.

[9] S. Borkar, T. Karnik, J. Tschanz, A. Keshavarzi, and V. De. Parameter variations and impact on circuits and microarchitecture. In *Proc. of 40th DAC*, 2003.

[10] F. A. Bower, D. J. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *Proc. of 38th MICRO*, 2005.

[11] 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. *J. of Solid-State Circuits*, 37(2):183–190, Feb 2002.

[12] D. Brooks and M. Martonosi. Dynamic thermal management for high-performance microprocessors. In *Proc. of 7th HPCA*, 2001.

[13] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proc. of 21st ICS*, 2007.

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

[15] C. Constantinescu. Intermittent faults in VLSI circuits. In *Proc. of Workshop on SELSE*, 2007.

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

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

[18] D. Ernst et al. Razor: A low-power pipeline based on circuit-level timing speculation. In *Proc. of 36th MICRO*, 2003.

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

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

[21] S. N. Hamilton and A. Orailoglu. Transient and intermittent fault recovery without rollback. In *Proc. of 13th Defect and Fault-Tolerance in VLSI Sys.*, 1998.

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

[23] R. Joseph. Exploring core salvage techniques for multi-core architectures. In *Proc. of Workshop on HPC Reliability Issues*, 2006.

[24] R. Joseph, D. Brooks, and M. Martonosi. Control techniques to eliminate voltage emergencies in high performance processors. In *Proc. of 9th HPCA*, 2003.

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

[26] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proc. of 13th PACT*, 2004.

[27] C. LaFrieda, E. Ípek, J. F. Martínez, and R. Manohar. Utilizing dynamically coupled cores to form a resilient chip multiprocessor. In *Proc. of 2007 DSN*, 2007.

[28] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *Trans. on Programming Languages and Sys.*, 4(3):382–401, 1982.

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

[30] X. Liang and D. Brooks. Mitigating the impact of process variations on processor register files and execution units. In *Proc. of 39th MICRO*, 2006.

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

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

[33] D. McEvoy. The architecture of tandem's nonstop system. In *Proc. of ACM 1981 Conf.*, 1981.

[34] S. Mitra, M. Zhang, N. S. amd TM Mak, and K. Kim. Soft error resilient system design through error correction. January 2006.

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

[36] 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 *Proc. of 11th ASPLOS*, 2004.

[37] M. D. Powell and T. N. Vijaykumar. Exploiting resonant behavior to reduce inductive noise. In *Proc. of 31st ISCA*, 2004.

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

[39] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *Proc. of 30th ISCA*, 2003.

[40] 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 *Proc. of 2002 DSN*, pages 389–398, Washington, DC, USA, 2002. IEEE Computer Society.

[41] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra low-cost defect protection for microprocessor pipelines. In *Proc. of 12th ASPLOS*, 2006.

[42] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proc. of 30th ISCA*, 2003.

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

[44] J. C. Smolens, B. T. Gold, J. C. Hoe, B. Falsafi, and K. Mai. Detecting emerging wearout faults. In *Proc. of Workshop on SELSE*, 2007.

[45] 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 *Proc. of 29th ISCA*, 2002.

[46] Sun Microsystems. Sun fire high-end and midrange systems dynamic reconfiguration user's guide. http://docs.sun.com/app/docs/doc/819-1501. Viewed 8/07/2007.

[47] 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. *J. of Solid-State Circuits*, 38(11), 2003.

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

[49] V. Uhlig, J. LeVasseur, E. Skoglund, and U. Dannowski. Towards scalable multiprocessor virtual machines. In *Proc. of 3rd Virt. Mach. Research and Tech. Symp.*, 2004.

[50] P. M. Wells, K. Chakraborty, and G. S. Sohi. Hardware support for spin management in overcommitted virtual machines. In *Proc. of 15th PACT*, 2006.

[51] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal streaming of shared memory. In *Proc. of 32nd ISCA*, 2005.